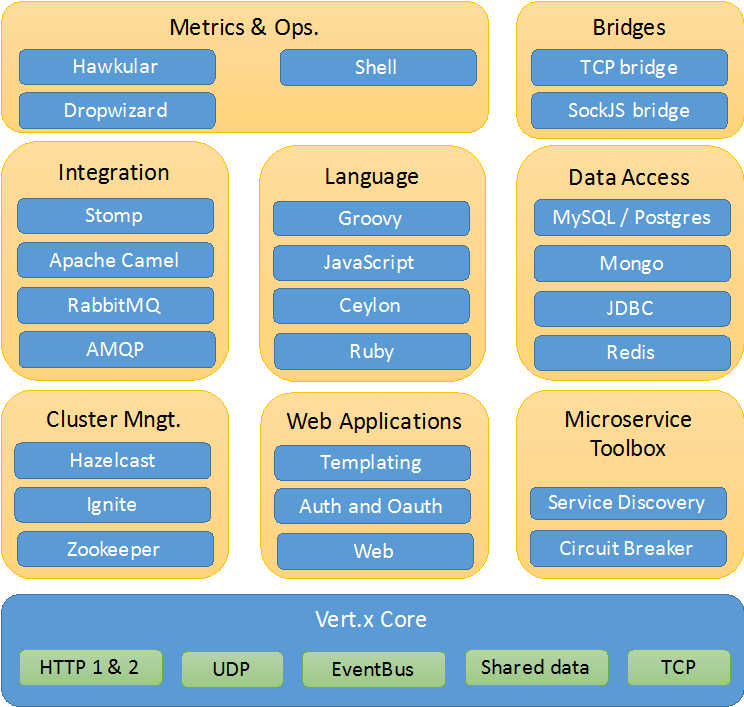
# Eclipse Vert.x

Vert.x是一个在JVM开发reactive应用的框架，可用于开发异步、可伸缩、高并发的Web应用，其目的在于为JVM提供一个Node.js的替代方案。开发者可以使用JavaScript、Ruby、Groovy、Java，甚至是混合语言来开发应用程序的组件。Vert.x的功能特性如下：

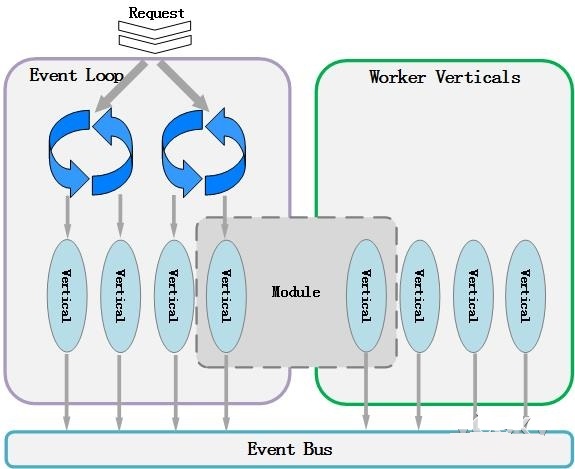
* Vert.x核心，包含底层次的功能，包括http、tcp、文件系统访问和各种其他功能，可以直接使用到应用中，同时被Vert.x其他组件使用
* Vert.x Web，编写复杂的Web应用和http微服务的工具箱
* 数据访问，Vert.x提供异步客户端，用于访问各数据源的数据
* Vert.x集成，邮箱客户端、STOMP客户端和服务器、JCA适配器、TCP Eventbus bridge
* 身份验证和授权，提供简单的API在应用中进行验证，同时提供盒子外的实现，常见的验证包括JDBC验证、JWT验证、Shiro验证、MongoDB验证、OAuth2
* Reactive，Vert.x提供一对组件让应用的Reactive性更好，其中包括Vert.x Rx、反应流、Sync等
* 微服务，Vert.x提供各种组件去创建创建基于微服务的应用，包括服务发现组件（Vert.x Consul）、Vert.x断路器、配置等
* Devops，在生产环境中使用Dropwizard运行Metric时，可以使用Hawkular、Shell、JMX等保证Vert.x应用正常
* 测试，Vert.x-Unit是一个单元测试工具包，特别是用来设计异步工作代码
* 云支持，Openshift2
* 高级特性，可以通过Hazelcast等分布式组件实现分布式Vert.x

其生态系统图如下所示：



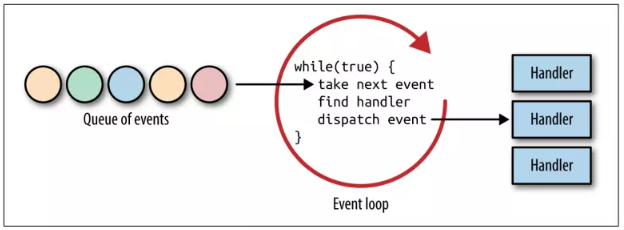
# **Vert.x基本概念**

Vert.x基于事件驱动模型，其处理请求的高性能也是基于事件机制，如下图：



Vert.x事件机制中的重要概念如下：

1）Event Loops，事件循环，由Vert.x启动的时间处理线程，Vert.x由Event Loop接收请求的事件，包括多个事件循环线程，其个数由主机的CPU核数决定。



1. Vert.x Thread，Event Loop及Worker均由Vert.x Thread类运行，并通过VertxThreadFactory线程工厂来创建
2. Vertical，事件的业务处理线程，包括Event Loop Vertical和Worker Vertical两种类型，基于Vert.x服务需要多Verticals来实现，并且部署在多台服务器中，之间通过Vert.x事件进行通信
3. Handler，Vertical对象包含一个或者多个处理器Handler，在Java中经常以Lambda即匿名函数的形式出现，如：

*vertx.createHttpServer().requestHandler(req->{*

*//blablabla*

*}).listen(8080);*

1. Module，项目模块，应用由多个模块组成，每个模块包含多个Vertical，其相当于公共的服务实现，开发者可以将开发好的Module发布到Maven仓库或者注册中心中
2. Event Bus，Vert.x的核心，用于在集群容器之间的通信，各Verticle之间的通信都是经过Event Bus来实现
3. Shared Data，提供简单共享的Map和Set，用来解决各Vertical之间的数据共享

Vert.x以非阻塞的IO思想来实现功性能，核心基于Event Loop Vertical和Worker Vertical分离。其处理流程如下：

* 在Vert.x中，Event Loop用于接收请求，并将短业务操作交由内部的Vertical来处理的，该模块是非阻塞的，以此保证请求的处理效率。
* 阻塞任务通过Vert.x的事件分离机制，生成任务事件交由Event Bus进行处理，其监听所有的事件，通过事件匹配选择合适的Worker Vertical，并将请求由Worker Vertical进行处理。
* Worker Vertical处理完成后，将数据和事件信息进行封装，返回给Event Bus，再次匹配事件来源Vertical，并将数据返回给Vertical
* Vertical将数据返回给客户端

Vert.x处理的核心是Event Bus，其管理所有的时间，通过事件匹配和数据传递，将整个流程耦合在一起。

# **Vert.x使用示例**

## 2.1 基本使用

创建Maven项目，项目结构如下所示：

*├── pom.xml*

*├── src*

*│   ├── main*

*│   │   ├── java*

*│   │   └── resources*

*│   └── test*

*│   └── java*

1. 添加pom依赖

*<dependencies>*

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-core</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*</dependencies>*

1. 定义Verticle

*public class MyFirstVerticle extends AbstractVerticle {*

*public void start() {*

*vertx.createHttpServer().requestHandler(*

*req -> {*

*req.response()*

*.putHeader("content-type", "text/plain")*

*.end("Hello World");*

*} ).listen(8080);}}*

定义测试类

*public class Main {*

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

*Vertx vertx = Vertx.vertx();*

*vertx.deployVerticle(MyFirstVerticle.class.getName());*

*}*

*}*

1. 编译及运行

在pom中添加编译相关依赖如下：

*<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.apache.maven.plugins</groupId>*

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

*<version>2.4.2</version>*

*<executions>*

*<execution>*

*<phase>package</phase>*

*<goals>*

*<goal>shade</goal>*

*</goals>*

*<configuration>*

*<transformers>*

*<transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">*

*<manifestEntries>*

*<Main-Class>${main.class}</Main-Class>*

*</manifestEntries>*

*</transformer>*

*</transformers>*

*<artifactSet />*

*<outputFile>${project.build.directory}/${project.artifactId}-${project.version}-prod.jar</outputFile>*

*</configuration>*

*</execution>*

*</executions>*

*</plugin>*

*</plugins>*

*</build>*

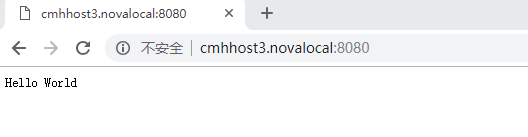
编译及执行

*mvn clean package -DskipTests*

执行命令如下：

*java -jar vert-example-1.0-SNAPSHOT-prod.jar*

运行后，在浏览器的地址栏中输入http://localhost:8080，则输出如下：



## **2.2 Event Bus**

Event Bus是Vert.x的神经系统，每个Vert.x实例都有一个单独的Event bus实例，可以通过Vert.x实例的eventBus方法来获得对应的EventBus实例。应用的不同部分通过Event Bus相互通信，无论在不同的Vert.x实例或者同一个Vert.x实例，甚至可以通过桥接的方式运行在浏览器中运行的客户端JavaScript在相同的Event Bus上相互通信。通过Event Bus可实现：

1. 可形成跨越多个服务器节点和多个浏览器的点对点的分布式消息系统
2. 支持发布/订阅、点对点、请求/响应的消息通信方式

Event Bus的API相对简单，基本上只涉及处理器的注册、撤销及消息的发送、发布等。

### 2.2.1 基本概念

下面是基本概念：

* 寻址，消息会被Event Bus发送到一个地址，Vert.x地址是一个简单的字符串、任意字符串都合法。合法的地址：

*europe.news.feed1、acme.games.pacman、sausages和X*

* 处理器，消息在处理器（Handler）中被接收，可以在某个地址上注册一个处理器来接收消息。同一个地址可以注册不同的处理器，一个处理器可以注册到多个不同的地址上。
* 发布/订阅消息，支持发布消息功能，消息将被发布到一个地址中，发布意味着会将消息传递给所有注册在该地址上的处理器
* 点对点/请求-响应模式，若地址上不止一个注册过的处理器，它将使用不严格的轮询算法选择其中一个，点对点消息传递模式下，可在消息发送的时候指定一个应答处理器。当接收者收到消息并且已经处理时，可以选择性决定回复该消息，若选择回复则绑定的应答处理器将会被调用。当发送者收到回复消息时，它也可以回复，这个过程可以不断重复。通过这种方式可以允许在两个不同的Verticle之间设置一个对话窗口，这种消息模式被称作请求-响应模式。
* 尽力传输，Vert.x会尽最大努力去传递消息，并且不会主动丢弃消息，这种方式称为尽力传输（Best-effort delivery），但是当Event Bus中的全部会部分发生故障时，则可能会丢失消息。对于RPC通信通常情况下有三种语义：at least once、at most once和exactly once。
* 消息类型，Vert.x默认允许任何基本/简单类型、String或者Buffer作为消息发送。不过在Vert.x中通常做法是使用JSON格式来发送消息。在Event Bus也支持发送任意对象，可以通过自定义MessageCodec来实现。

### **2.2.2 使用**

下面介绍Event-Bus的使用

1. 定义Sender类

*public class Sender extends AbstractVerticle {*

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

*Runner.runClusteredExample(Sender.class);*

*}*

*@Override*

*public void start() throws Exception {*

*EventBus bus = vertx.eventBus();*

*System.out.println("Sender Ready, Start to receive reply!");*

*vertx.setPeriodic(100, v -> {*

*bus.send("ping-address", "ping!", reply -> {*

*if(reply.succeeded()) {*

*System.out.println("Received reply " + reply.result().body());*

*} else {*

*System.out.println("No reply");*

*}*

*});*

*});*

*}}*

1. 定义Receiver类

*public class Receiver extends AbstractVerticle {*

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

*Runner.runClusteredExample(Receiver.class);*

*}*

*@Override*

*public void start() throws Exception {*

*EventBus bus = vertx.eventBus();*

*bus.consumer("ping-address", message -> {*

*System.out.println("Received message: " + message.body());*

*message.reply("pong!");*

*});*

*System.out.println("Receiver ready!");*

*}*

*}*

1. 在pom.xml中添加依赖

*<dependencies>*

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-core</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*<dependency> //集群模式运行Event Bus*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-hazelcast</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*<dependency>*

*<groupId>io.netty</groupId>*

*<artifactId>netty-tcnative-boringssl-static</artifactId>*

*<version>2.0.7.Final</version>*

*</dependency>*

*</dependencies>*

1. 执行如下

Sender执行输出如下：

*Received reply pong!*

*Received reply pong!*

*Received reply pong!*

*.......*

Receiver执行输出如下：

*Received message: ping!*

*Received message: ping!*

*Received message: ping!*

*......*

### **2.2.3 Java API**

1) 获取Event Bus

*EventBus eb = vertx.eventBus();*

2）注册处理器

最简单的注册处理器的方式是使用consumer方法，示例如下：

*eventBus.consumer("cluster-message-receiver", message -> {*

*CustomMessage customMessage = (CustomMessage) message.body();*

*System.out.println("Custom message received: " + customMessage.getSummary());*

*CustomMessage replyMessage = new CustomMessage(200, "a00000002","Message sent from cluster receiver!");*

*message.reply(replyMessage);*

*});*

当一个消息达到处理器，该处理器会以message为参数被调用，调用consumer方法会返回MessageConsumer对象，该对象随后可用于撤销处理器或将处理器用作流式处理。

在MessageConsumer对象上注册handler，示例如下：

*MessageConsumer<String> consumer = eb.consumer("news.uk.sport");*

*consumer.handler(message -> {*

*System.out.println("I have received a message: " + message.body());*

*});*

1. 撤销处理器，可以通过unregister方法来撤销处理器，示例如下：

*consumer.unregister(res -> {*

*if (res.succeeded()) {*

*System.out.println("The handler un-registration has reached all nodes");*

*} else {*

*System.out.println("Un-registration failed!");*

*}*

*});*

1. 发布消息，使用publish方法指定一个地址发布即可

*eventBus.publish("news.uk.sport", "Yay! Someone kicked a ball");*

这个消息将会传递给所在地址news.uk.sport上注册的处理器。

1. 发送消息，send消息只会传递给在该地址注册的其中一个处理器，这就是点对点模式，Vert.x使用不严格的轮询算法来选择绑定的处理器

*eventBus.send("news.uk.sport", "Yay! Someone kicked a ball");*

### **2.2.4 消息编解码器**

可以在Event Bus中发送任何对象，为这个对象类型注册一个编解码器MessageCodec。消息编解码有一个名词，需要在发送或者发送消息时通过DeliveryOptions来指定：

*EventBus eventBus = getVertx().eventBus();*

*// Register codec for custom message*

*eventBus.registerDefaultCodec(CustomMessage.class, new CustomMessageCodec());*

也可以通过unregisterCode方法注销某个消息编解码器。下面示例是编写编解码器来发送POJO对象，但是当消息发送给处理器后解码成OtherPOJO对象。

1. CustomMessage定义

*public class CustomMessage {*

*private final int statusCode;*

*private final String resultCode;*

*private final String summary;*

*.....*

*}*

1. 定义CustomMessageCodec

*public class CustomMessageCodec implements MessageCodec<CustomMessage, CustomMessage> {*

*@Override*

*public void encodeToWire(Buffer buffer, CustomMessage customMessage) {*

*// Easiest ways is using JSON object*

*JsonObject jsonToEncode = new JsonObject();*

*jsonToEncode.put("statusCode", customMessage.getStatusCode());*

*jsonToEncode.put("resultCode", customMessage.getResultCode());*

*jsonToEncode.put("summary", customMessage.getSummary());*

*// Encode object to string*

*String jsonToStr = jsonToEncode.encode();*

*// Length of JSON: is NOT characters count*

*int length = jsonToStr.getBytes().length;*

*// Write data into given buffer*

*buffer.appendInt(length);*

*buffer.appendString(jsonToStr);*

*}*

*@Override*

*public CustomMessage decodeFromWire(int position, Buffer buffer) {*

*// My custom message starting from this \*position\* of buffer*

*int \_pos = position;*

*// Length of JSON*

*int length = buffer.getInt(\_pos);*

*// Get JSON string by it`s length*

*// Jump 4 because getInt() == 4 bytes*

*String jsonStr = buffer.getString(\_pos+=4, \_pos+=length);*

*JsonObject contentJson = new JsonObject(jsonStr);*

*// Get fields*

*int statusCode = contentJson.getInteger("statusCode");*

*String resultCode = contentJson.getString("resultCode");*

*String summary = contentJson.getString("summary");*

*// We can finally create custom message object*

*return new CustomMessage(statusCode, resultCode, summary);*

*}*

*@Override*

*public CustomMessage transform(CustomMessage customMessage) {*

*// If a message is sent \*locally\* across the event bus.*

*// This example sends message just as is*

*return customMessage;*

*}*

*@Override*

*public String name() {*

*// Each codec must have a unique name.*

*// This is used to identify a codec when sending a message and for unregistering codecs.*

*return this.getClass().getSimpleName();*

*}*

*@Override*

*public byte systemCodecID() {*

*// Always -1*

*return -1;*

*}*

*}*

## **2.3 集群模式的Vert.x**

通过编程的方式创建Vert.x实例，通过Vert.x实例配置成集群模式来获取集群模式的Event Bus，示例如下：

*VertxOptions options = new VertxOptions();*

*Vertx.clusteredVertx(options, res -> {*

*if (res.succeeded()) {*

*Vertx vertx = res.result();*

*EventBus eventBus = vertx.eventBus();*

*System.out.println("We now have a clustered event bus: " + eventBus);*

*} else {*

*System.out.println("Failed: " + res.cause());*

*}*

*});*

在classpath中包含ClusterManager的实现类，默认的HazelcastClusterManager，如在pom中添加依赖如下:

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-hazelcast</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

https://vertxchina.github.io/vertx-translation-chinese/core/Core.html

## **2.3 TCP客户端和服务端**

可以通过Vert.x编写非阻塞的TCP客户端和服务器，Server端实现示例如下：

*vertx.createNetServer().connectHandler( //定义Handler*

*sock -> { //客户端连接后，会从Socket中读取数据*

*Pump.pump(sock, sock).start();*

*}).listen(1234); //监听1234端口*

客户端实现示例如下：

*vertx.createNetClient().connect(1234, "cmhhost3.novalocal", //连接Server*

*res -> {*

*if (res.succeeded()) {*

*NetSocket socket = res.result(); //获取Socket*

*socket.handler(*

*buffer -> { //Socket接收到数据时，以Buffer对象为参数调用处理器*

*System.out.println("Net Client receiving: " + buffer.toString("UTF-8"));*

*}*

*);*

*for( int i = 0 ;i < 10 ;i ++) {*

*String str = "hello "+ i + "\n";*

*System.out.println("Net Client Sending: " + str);*

*socket.write(str); //向Socket写入数据*

*}*

*} else {*

*System.out.println("Failed to connect " + res.cause());*

*}*

*});*

## **2.4 HTTP服务端和客户端**

Vert.x提供Http客户端和服务端的API，其支持HTTP/1.0,1.1及HTTP/2协议，创建HTTP服务端的示例如下：

*vertx.createHttpServer().requestHandler(*

*req -> { // HttpServerRequest*

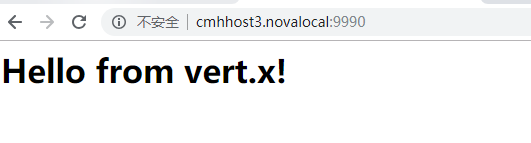
*req.response().putHeader("content-type","text/html")*

*.end("<html><body><h1>Hello from vert.x!</h1></body></html>");*

*}*

*).listen(9990); //监听端口*

在浏览器中测试如下：



## **2.5共享数据**

Shared Data包含的功能允许用户安全的在应用程序的不同部分之间，同一Vert.x实例中不同应用程序之间或者集群中不同Vert.x实例之间安全的共享数据，包括本地共享Map、分布式、集群范围Map、异步集群范围锁和异步集群范围计数器。分布式数据结构的行为取决于其使用的集群管理器，网络分区面临的复制和行为由集群管理器和它的配置来定义。

可以使用getClusterWideApp方法获取AsyncMap实例，其使用如下：

*SharedData sd = vertx.sharedData();*

*sd.<String,String>getClusterWideMap("mymap", res -> {*

*if(res.succeeded()) {*

*AsyncMap<String,String> mymap = res.result(); //获取AsyncMap实例*

*mymap.put("foo","value", result -> { //放入数据*

*if(result.succeeded()) {*

*System.out.println("Data puted successed!1");*

*}*

*});*

*} else {*

*System.out.println("Something went wrong!");*

*}*

*});*

从Map中读取数据，get方法也是异步的，其执行如下：

*SharedData sd = vertx.sharedData();*

*sd.<String,String>getClusterWideMap("mymap", res -> {*

*if(res.succeeded()) {*

*AsyncMap<String,String> myMap = res.result();*

*map.get("foo", resGet -> { //从Map中读取数据*

*Object value = resGet.result();*

*})*

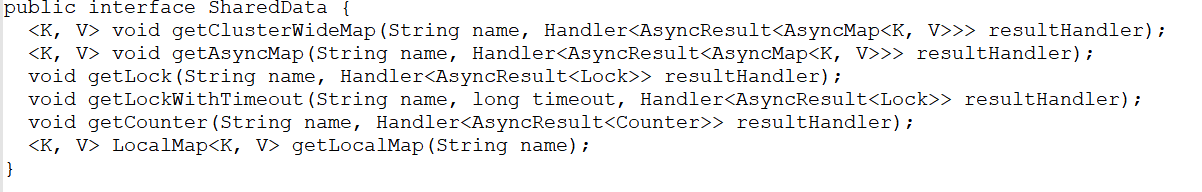
*} else {*

*System.out.println("Something went wrong !!!");*

*}*

*});*

还支持其他的操作，如下所示：



## **2.6命令行**

Vert.x命令行工具用于在终端中与Vert.x进行交互，主要用于运行Vert.x Verticle

1）下载Vert.x安装包

*https://bintray.com/artifact/download/vertx/downloads/vert.x-3.5.4-full.tar.gz*

2）解压并安装

1. 运行命令 //将依赖包添加到classpath中

*# bin/vertx run com.fys.example.MyFirstVerticle*

*Succeeded in deploying verticle*

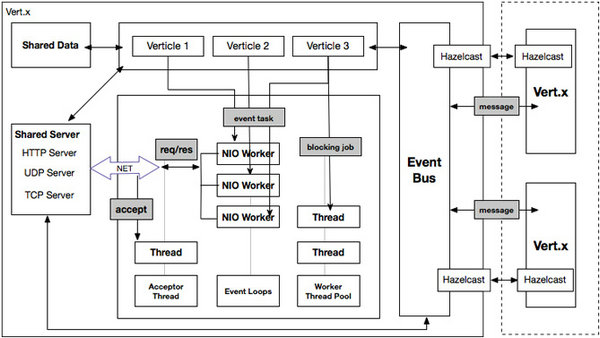
还支持以下命令的执行

# **Vert.x源码分析**

Vert.x是一个用于构建reactive和分布式系统的工具箱，其内部通过Netty来实现异步非阻塞编程模型，在同一个集群中，各实例是消息驱动架构，通过Event Bus通信。本处介绍其Vert.x Core的实现实现。

## **Vertx系统架构**

Vertx的内部整体架构图如下所示：



其组成部分包括：

1）Verticle，业务逻辑都是基于Verticle实现，可以看做微服务，分成两种：

* 基于EventLoop的适合I/O密集型的，其绑定在Nio Worker上
* Worker Verticle，适合CPU密集型，主要用来处理同步信息，退化到传统基于多线程模型的实现

Verticle之间通过EventBus来相互交互。

2）Shared Server，包括HttpServer ,Tcp Server，UDP Server等，将Server部署在Verticle中，Server的连接监听在Acceptor EventLoop中

3）Acceptor Thread，通过Acceptor EventLoop监听所有客户端连接，建立后交给Event Loop，每一个Verticle都要绑定Event Loop，其实现为VertxEventLoopGroup

4）Context，每部署Verticle都会创建一个执行Context（对应线程或者执行器），业务Request Handler都在此Context对应的线程或执行器中执行

5）EventBus，应用的不同部分通过Event Bus相互通信，无论在不同的Vert.x实例或者同一个Vert.x实例，甚至可以通过桥接的方式运行在浏览器中运行的客户端JavaScript在相同的Event Bus上相互通信。

6）ClusterManager，集群管理器，默认Hazelcast实现，发现和编组集群中Vert.x节点成员

## **Vertical**

要使用Vertx的第一步就是创建Vertx对象，其初始化如下：

*Vertx vertx = Vertx.vertx();*

Verticles是Vertx的一个模型，提供给开发者封装代码，可以把Verticles当做Actor模型中的Actor。Vertx的服务由多个Verticle来实现。Verticle的实现继承AbstractVerticle，类图如下所示：



下面是实现类：

*public class BasicVerticle extends AbstractVerticle {*

*@Override*

*public void start() throws Exception {*

*System.out.println("BasicVerticle started");*

*}*

*}*

创建Verticle后，就可以注册到Vert.x实例中，其执行如下：

*Vertx vertx = Vertx.vertx();*

*vertx.deployVerticle(new BasicVerticle());*

Vert.x的类图如下所示：



Verticle部署如下：



调用如下：

*private void doDeploy(String identifier, String deploymentID, DeploymentOptions options,*

*ContextInternal parentContext,*

*ContextInternal callingContext,*

*Handler<AsyncResult<String>> completionHandler,*

*ClassLoader tccl, Verticle... verticles) {*

*Deployment parent = parentContext.getDeployment();*

*DeploymentImpl deployment = new DeploymentImpl(parent, deploymentID, identifier, options);*

*.....*

*for (Verticle verticle: verticles) {*

*WorkerExecutorInternal workerExec = poolName != null ? vertx.createSharedWorkerExecutor(poolName, options.getWorkerPoolSize(), options.getMaxWorkerExecuteTime(), options.getMaxWorkerExecuteTimeUnit()) : null;*

*WorkerPool pool = workerExec != null ? workerExec.getPool() : null;*

*ContextImpl context = options.isWorker() ? vertx.createWorkerContext(options.isMultiThreaded(), deploymentID, pool, conf, tccl) :*

*vertx.createEventLoopContext(deploymentID, pool, conf, tccl);*

*context.setDeployment(deployment);*

*deployment.addVerticle(new VerticleHolder(verticle, context));*

*context.runOnContext(v -> {*

*try {*

*verticle.init(vertx, context);*

*Future<Void> startFuture = Future.future();*

*verticle.start(startFuture);*

*......*

*deployments.put(deploymentID, deployment);*

*}*

*}*

部署后，调用Verticle#start()，Vertcile有三种类型的Verticle

1. **Standard Verticles**

标准的Verticle会被分派到event loop线程上，该Standard Verticle中注册的所有回调handler都会被这个event loop线程来调用，无论有多少次请求都会被该线程来处理。

*Verticle myVerticle = new MyVerticle();*

*vertx.deployVerticle(myVerticle);*

1. **Worker Verticles**

Worker Verticle会被Worker thread pool线程池中的一个线程来执行，执行之后，Worker Verticle不会从属于这个worker thread。

*DeploymentOptions options = new DeploymentOptions().setWorker(true); vertx.deployVerticle(new MyVerticle(), options);*

1. **Multi-threaded worker verticles，多线程Verticles**

Standard Verticle从属于一个Event Loop线程，无论多少次请求处理，都只会被这个event Loop线程来处理。当请求超过event loop处理会造成线程阻塞，导致异常发生。和Standard Verticle相比，Worker verticles不会阻塞event loop线程。

## **Vert.x线程模型**

Vert.x有两种线程：Event Loop线程和Worker线程，其执行的核心Vert.x Thread及Vert.x Context，不同的线程通过Context来调用，如下所示：



*ContextImpl context = options.isWorker() ? vertx.createWorkerContext(options.isMultiThreaded(), deploymentID, pool, conf, tccl) :*

*vertx.createEventLoopContext(deploymentID, pool, conf, tccl);*

*context.setDeployment(deployment);*

*deployment.addVerticle(new VerticleHolder(verticle, context));*

*context.runOnContext(v -> {*

*verticle.init(vertx, context);*

*Future<Void> startFuture = Future.future();*

*verticle.start(startFuture);*

*startFuture.setHandler(ar -> {*

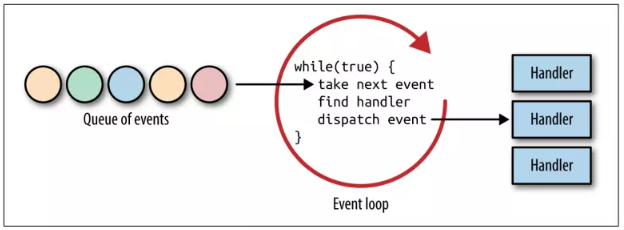
*.....*

*}}*

Vert.x中的Event Loop线程及Worker线程都用VertxThread类来表示，并通过ThreadFactory创建。

### **3.3.1 Event Loop线程**

Event Loop线程结合了Netty的EventLoop，用于处理事件，每一个EventLoop都与唯一的线程相绑定，这个线程叫做Event Loop，其不断轮询获取事件，并将获取到的事件分发到对应的事件处理器进行处理，如下图：



其使用如下所示：

*void executeAsync(Handler<Void> task) {*

*nettyEventLoop().execute(() -> executeTask(null, task));*

*}*

EventLoop的实例化在VertxImpl初始化时执行：

*eventLoopGroup = transport.eventLoopGroup(options.getEventLoopPoolSize(), eventLoopThreadFactory, NETTY\_IO\_RATIO);*

调用在ContextImpl#getEventLoop中，执行如下：

*private static EventLoop getEventLoop(VertxInternal vertx) {*

*EventLoopGroup group = vertx.getEventLoopGroup();*

*if (group != null) {*

*return group.next();*

*} ......*

*}*

在Event Loop中，不断轮询事件，并将获取到的事件分发到对应的事件处理器中进行处理，其执行如下：

*protected void run() {*

*for (;;) {*

*Runnable task = takeTask();*

*if (task != null) {*

*task.run();*

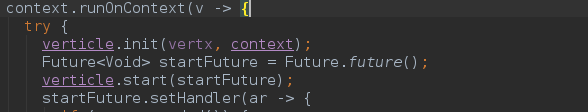
*updateLastExecutionTime();*

*}*

*} ......}*

该Task的形式是在deployVerticle时生成，其执行序列如下：

1. 将Verticle的执行在DeploymentManager中封装成task:Handler<Void>



1. EvengLoopContext中将task:Handler<void>封装成Task并执行

*ContextImpl#wrapTask*

*protected Runnable wrapTask(ContextTask cTask, Handler<Void> hTask, ...) {*

*return () -> {*

*Thread th = Thread.currentThread();*

*VertxThread current = (VertxThread) th;*

*.....*

*try {*

*setContext(current, ContextImpl.this);*

*if (cTask != null) {*

*cTask.run();*

*} else {*

*hTask.handle(null);*

*}*

*}*

*};*

EventLoop执行时将task添加到队列中，执行如下SingleThreadEventExecutor#execute：

*public void execute(Runnable task) {*

*....*

*this.addTask(task);*

*}*

这里启动了Verticle中的start方法。

### **3.3.2 WorkerContext**

对于WorkerContext，通过WorkerPool来调用，对外EventLoopContext则通过EventLoop执行Handler，WorkerContext的执行如下：

*public void executeAsync(Handler<Void> task) {*

*orderedTasks.execute(wrapTask(null, task, true, workerPool.metrics()), workerPool.executor()); //Executor#run*

*}*

Worker Context用于运行阻塞任务，与EventLoopContext相似，每个Handler都只运行在Worker线程下，其执行如下：

*public void execute(Runnable task, Executor executor) {*

*synchronized (tasks) {*

*tasks.add(new Task(task, executor));*

*if (current == null) {*

*current = executor;*

*executor.execute(runner);*

*}*

*}*

*}*

### **3.3.3 Handler**

根据Netty的模型，Event Loop线程需要处理IO事件、普通事件（Handler）以及定时事件。Vert.x会提供NETTY\_IO\_RATIO给Netty处理的IO事件时间占用的百分比，当EventLoop启动时，会不断轮询IO事件及其他事件并进行处理。

在部署Verticle时，会根据配置创建Context并绑定到Verticle上，该Verticle绑定的Handler都会在该Context上执行，Context对应EventLoop线程，而EventLoop线程对应多个Container。以HttpServer为例，其创建过程如下：

*server.requestHandler(req -> {*

*req.response().putHeader("content-type", "text/html").end("<html><body><h1>Hello from vert.x!</h1></body></html>");*

*}).listen(4443);*

requestHandler的执行时将Handler注册到requestStream中，如下所示：

*public synchronized HttpServer requestHandler(Handler<HttpServerRequest> handler) {*

*this.requestStream.handler(handler);*

*return this;*

*}*

当启动后EventLoop轮询事件，并交由Handler进行处理，HttpServer的启动过程：

*public synchronized HttpServer listen(SocketAddress address, Handler<AsyncResult<HttpServer>> listenHandler) {*

*.....*

*String host = address.host() != null ? address.host() : "localhost";*

*int port = address.port();*

*synchronized (vertx.sharedHttpServers()) {*

*if (shared == null || port == 0) {*

*serverChannelGroup = new DefaultChannelGroup("vertx-acceptor-channels", GlobalEventExecutor.INSTANCE);*

*ServerBootstrap bootstrap = new ServerBootstrap();*

*bootstrap.group(vertx.getAcceptorEventLoopGroup(), availableWorkers);*

*....*

*bootstrap.childHandler(new ChannelInitializer<Channel>() {*

*@Override*

*protected void initChannel(Channel ch) throws Exception {*

*if (!requestStream.accept() || !wsStream.accept()) {*

*ch.close();*

*return;*

*}*

*ChannelPipeline pipeline = ch.pipeline();*

*addHandlers(this, listenContext);*

*}*

*return this;*

*}*

addHandlers中添加Handler，如下：

*private void addHandlers(HttpServerImpl server, ContextInternal context) {*

*server.httpHandlerMgr.addHandler(*

*new HttpHandlers(*

*requestStream.handler(),*

*wsStream.handler(),*

*connectionHandler,*

*exceptionHandler == null ? DEFAULT\_EXCEPTION\_HANDLER : exceptionHandler)*

*, context);*

*}*

HttpServer通过netty ServerBootstrap创建，其是acceptor是单线程执行，在VertxImpl中定义：

*acceptorEventLoopGroup = transport.eventLoopGroup(1, //线程数目*

*acceptorEventLoopThreadFactory, 100); //ioRatio, 100%*

Netty的IO Worker线程由availableWorkers确定，其是VertxEventLoopGroup对象，其扩展了EventLoopGroup接口，

*eventLoopGroup = transport.eventLoopGroup(options.getEventLoopPoolSize(),*

*//poolSize <= 2 \* CpuCoreSensor.availableProcessors()*

*eventLoopThreadFactory, NETTY\_IO\_RATIO); //vertx.nettyIORatio，默认50%*

线程数量由Worker数量决定的。

https://www.sczyh30.com/posts/Vert-x/vertx-advanced-demystifying-thread-model/

## **EventBus**

Event Bus是Vert.x的神经系统，每个Vert.x实例都有单一Eveng Bus，其初始化如下：

*EventBus bus = vertx.eventBus();*

其核心方法是:

* consumer，注册处理事件，示例如下：

*eb.consumer("news.uk.sport", message -> {*

*System.out.println("receive a message: " + message.body());*

*});*

返回MessageConsumer对象实例，处理数据流。

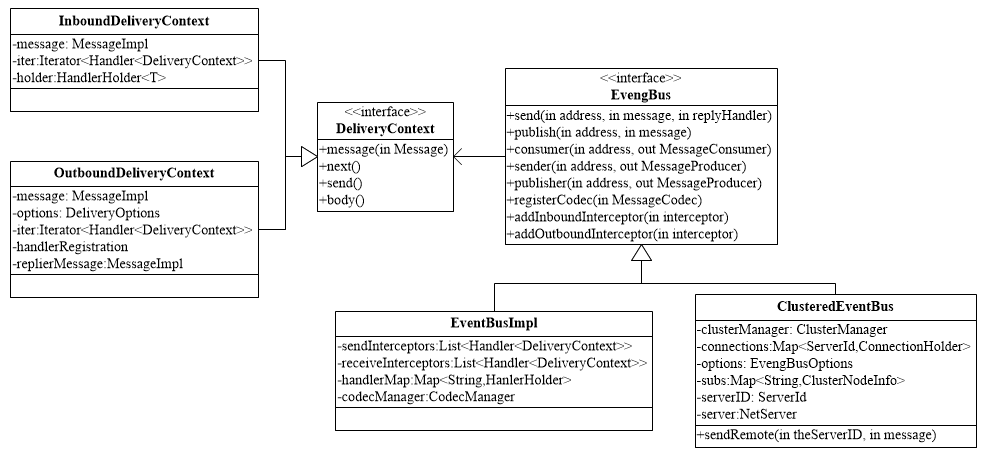
* send，发送消息，仅有一个注册地址会接收到消息

*eventBus.send("news.uk.sport", "Yay! Someone kicked a ball");*

* publish，公布消息，公布到指定地址

*eventBus.publish("news.uk.sport", "Yay! Someone kicked a ball");*

将消息交付给全部订阅news.uk.sport地址处理。Eveng Bus类图如下所示：



Vertx.eventBus的实现如下所示：

*if (options.isClustered()) { //集群化EventBus*

*this.clusterManager = getClusterManager(options);*

*this.eventBus = new ClusteredEventBus(this, options, clusterManager);*

*} else {*

*this.clusterManager = null; //Local EventBus*

*this.eventBus = new EventBusImpl(this);*

*}*

### **Local模式EventBusImpl**

1. **consume**，注册Consumer到Event Bus，其原型位于EventBus接口中：

*<T> MessageConsumer<T> consumer(String address);*

*<T> MessageConsumer<T> consumer(String address, Handler<Message<T>> handler);*

其实现位于EventBusImpl类中：

*public <T> MessageConsumer<T> consumer(String address, Handler<Message<T>> handler) {*

*Objects.requireNonNull(handler, "handler"); //要确保传入的handler不为空*

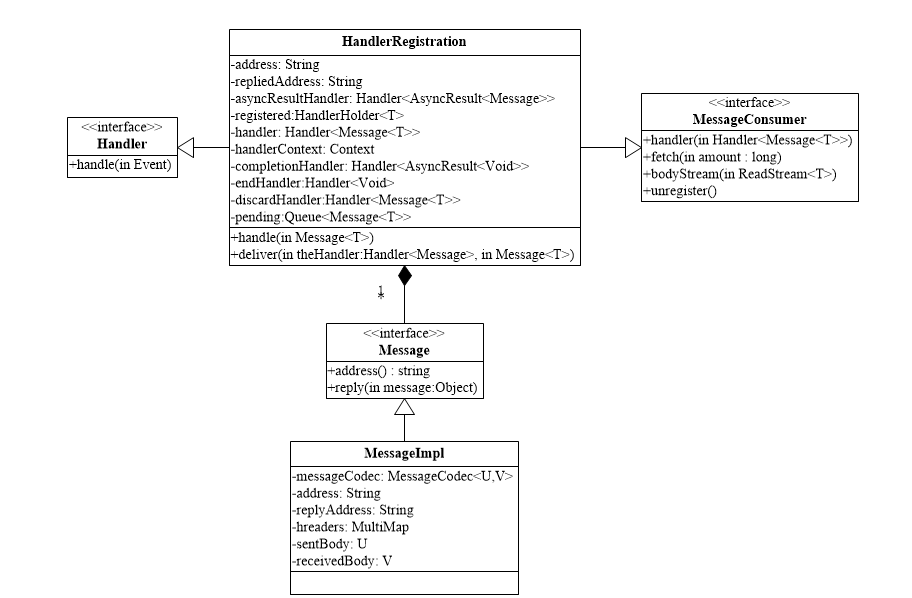
*MessageConsumer<T> consumer = consumer(address);*

*consumer.handler(handler); //将初始化的MessageConsumer绑定传入的handler*

*return consumer;*

*}*

MessageConsumer相关的类图如下所示：



HandlerRegistration类同时继承了MessageConsumer<T>和Handler<Message<T>>接口，从其类名可以看到相当于Handler注册记录。其调用handler方法中将handler成员设置为传入的消息处理函数，执行如下：

*public synchronized MessageConsumer<T> handler(Handler<Message<T>> handler) {*

*this.handler = handler;*

*if (this.handler != null && registered == null) {*

*registered = eventBus.addRegistration(address, this, repliedAddress != null, localOnly);*

*} //将handler注册到EventBus中*

*return this;}*

addRegistration方法接收四个参数：

* address，发送地址
* replyHandler，代表是否为replyHandler
* localOnly，是否在集群范围内传播的标志位
* handlerRegistration

将HandlerRegistration添加到handlerMap:Map<String,ConcurrentCyclicSeqe<HandlerHolder>>中，key为地址，Value为对应的HandlerHolder集合，其内部维护list列表。Handlers类中有一个Choose函数，根据轮询算法从HandlerHolder集合中选定一个Handler，

*ConcurrentCyclicSequence#next*

*public T next() {*

*int len = elements.length;*

*switch (len) {*

*case 0:*

*return null;*

*case 1:*

*return (T) elements[0];*

*default:*

*int p;*

*p = pos.getAndIncrement();*

*return (T) elements[Math.abs(p % len)];*

*}*

*}*

1. **send & publish**

EventBusImpl#send方法执行如下：

*eventBus.send("news.uk.sport", "Yay! Someone kicked a ball");*

send方法是发送至目标地址为”news.uk.sport”的某个消费者。pubish是发布至所有消费者。 下面介绍send方法的分析，最终都归结于生成消息然后调用sendOrPubInternal方法执行逻辑，执行如下：

*sendOrPubInternal(*

*createMessage(true, address, options.getHeaders(), message, options.getCodecName()),*

*options,*

*replyHandler);*

1. 封装MessageImpl

createMessage方法接收5个参数：

* send，标志位
* address，发送目标地址
* headers，为设置的header
* body，代表发送的对象
* codeName，代表对象的Codec名称，消息编码解码器

*protected MessageImpl createMessage(boolean send, String address, MultiMap headers, Object body, String codecName) {*

*Objects.requireNonNull(address, "no null address accepted");*

*MessageCodec codec = codecManager.lookupCodec(body, codecName);*

*//获取MessageCodec，将body封装成MessageImpl，其中包含发送的地址*

*MessageImpl msg = new MessageImpl(address, null, headers, body, codec, send, this);*

*return msg;*

*}*

MessageImpl的构造函数如下：

*public MessageImpl(String address, String replyAddress, MultiMap headers, U sentBody,*

*MessageCodec<U, V> messageCodec,*

*boolean send, EventBusImpl bus) {*

*this.messageCodec = messageCodec; // Codec*

*this.address = address; // 发送目标地址*

*this.replyAddress = replyAddress; // 回复地址*

*this.headers = headers; // header*

*this.sentBody = sentBody; // 发送的对象*

*this.send = send; // 是否为点对点模式*

*this.bus = bus; // 相关的Event Bus实例*

*}*

b）发送MessageImpl，其通过sendOrPubInternal方法，其执行过程如下：

*private <T> void sendOrPubInternal(*

*MessageImpl message, //要发送的消息*

*DeliveryOptions options, //发送配置选项*

*Handler<AsyncResult<Message<T>>> replyHandler //回复处理函数*

*) {*

*checkStarted(); //检查Vertx是否启动*

*HandlerRegistration<T> replyHandlerRegistration =*

*//包装成回复地址，并生成对应reply consumer*

*createReplyHandlerRegistration(message, options, replyHandler);*

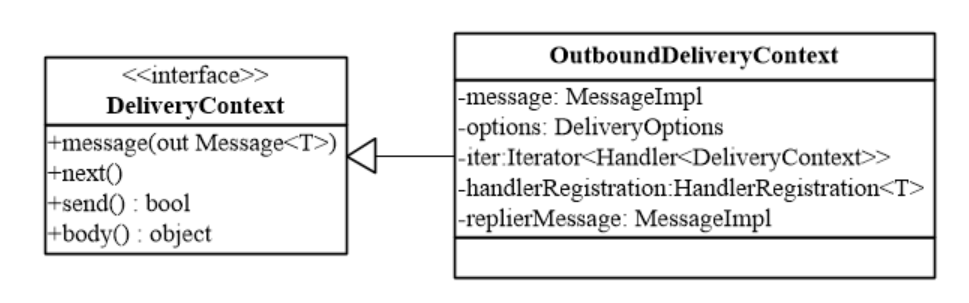
*OutboundDeliveryContext<T> sendContext =*

*new OutboundDeliveryContext<>(message, options, replyHandlerRegistration);*

*sendContext.next();*

*}*

Vert.x会创建OutboundDeliveryContext实例，并调用next方法，其类实现了DeliveryContext接口，相当于消息的封装体，并且可以与Event Bus中的interceptors结合使用，其类图如下所示：



DeliveryContext接口中定义了三个方法：

* + message，获取当前SendContext包装的消息实体
  + next，调用下一个消息拦截器
  + send，代表消息的发送模式是否为点对点模式

在Event Bus中，消息拦截器本质上是Handler<SendContext>类型的处理函数，在其内部存储着interceptors:List<Handler<DeliveryContext>> sendInterceptors列表，用于存储绑定的消息拦截器，其可以通过add和remove来实现。

c）消息的发送

Event Bus通过DeliveryContext#next方法来实现消息发送

*public void next() {*

*if (iter.hasNext()) {*

*Handler<DeliveryContext> handler = iter.next(); //每次调用都是通过拦截器进行调用*

*try {*

*if (handler != null) {*

*handler.handle(this);*

*} else {*

*next();*

*}*

*} catch (Throwable t) {*

*log.error("Failure in interceptor", t);*

*}*

*} else { //无拦截器后，调用sendReply发送消息，最终调用delierMessageLocally*

*if (replierMessage == null) {*

*sendOrPub(this);*

*} else {*

*sendReply(this, replierMessage);*

*}*

*}*

*}*

最终调用deliverMessageLocally方法，此方法返回值代表发送消息的目的地址是否注册有MessageConsumer，区分send和publish方法，其实现如下：

*protected <T> boolean deliverMessageLocally(MessageImpl msg) {*

*msg.setBus(this);*

*//查询发送消息的目的地址是否注册有MessageConsumer*

*ConcurrentCyclicSequence<HandlerHolder> handlers = handlerMap.get(msg.address());*

*if (handlers != null) {*

*if (msg.isSend()) { //Send方法*

*//Choose one*

*HandlerHolder holder = handlers.next();*

*.....*

*if (holder != null) {*

*deliverToHandler(msg, holder); //执行Server注册的Consumer，其Handler*

*}*

*} else { // Publish方法*

*for (HandlerHolder holder: handlers) {*

*deliverToHandler(msg, holder);*

*}*

*}*

*return true;*

*}*

deliveryToHandler方法会从传入的HandlerHolder中获取Vert.x Context，调用runOnContext方法让消息处理逻辑在Context中执行。

*private <T> void deliverToHandler(MessageImpl msg, HandlerHolder<T> holder) {*

*MessageImpl copied = msg.copyBeforeReceive();*

*DeliveryContext<T> receiveContext = new InboundDeliveryContext<>(copied, holder);*

*holder.getContext().runOnContext((v) -> {*

*try {*

*if (!holder.isRemoved()) {*

*receiveContext.next();*

*}*

*} finally {*

*if (holder.isReplyHandler()) { //如果是reply Consumer，则自动注销该Consumer*

*holder.getHandler().unregister();*

*}*

*}*

*});*

*}*

HandlerRegistration类继承了ReadStream接口，定义了两个函数用于控制Stream的通断。当处理速度小于读取速度的时候可以通过pause方法暂停消息的传递，将积压的消息暂存与内部的消息队列的pending中。

d) 消息的处理，通过MessageConsumer#handle

*public void handle(Message<T> message) {*

*Handler<Message<T>> theHandler;*

*synchronized (this) {*

*if (demand == 0L) {*

*if (pending.size() < maxBufferedMessages) {*

*pending.add(message); //添加到pending队列*

*} ...*

*deliver(theHandler, message); //真正处理消息逻辑的地方*

*}*

deliver方法是真正执行消息处理逻辑的地方

*private void deliver(Handler<Message<T>> theHandler, Message<T> message) {*

*checkNextTick(); //检查消息队列中是否有更多的消息，*

*//取出消息并调用deliver方法传给handler处理*

*String creditsAddress = message.headers().get(MessageProducerImpl.CREDIT\_ADDRESS\_HEADER\_NAME);*

*if (creditsAddress != null) {*

*eventBus.send(creditsAddress, 1); //如何creditsAddress，则向地址发送消息*

*}*

*theHandler.handle(message); //处理消息*

*}*

*}*

e) 在HandlerRegistration的handle处理逻辑中，调用Message的reply方法，如下：

*eventBus.consumer("local-message-receiver", message -> {*

*CustomMessage customMessage = (CustomMessage) message.body();*

*CustomMessage replyMessage = new CustomMessage(200, "a00000002","Messae sent from local receiver!");*

*message.reply(replyMessage);*

*});*

其实现方法位于MessageImpl中，其执行如下：

*public <R> void reply(Object message, DeliveryOptions options,*

*Handler<AsyncResult<Message<R>>> replyHandler) {*

*if (replyAddress != null) {*

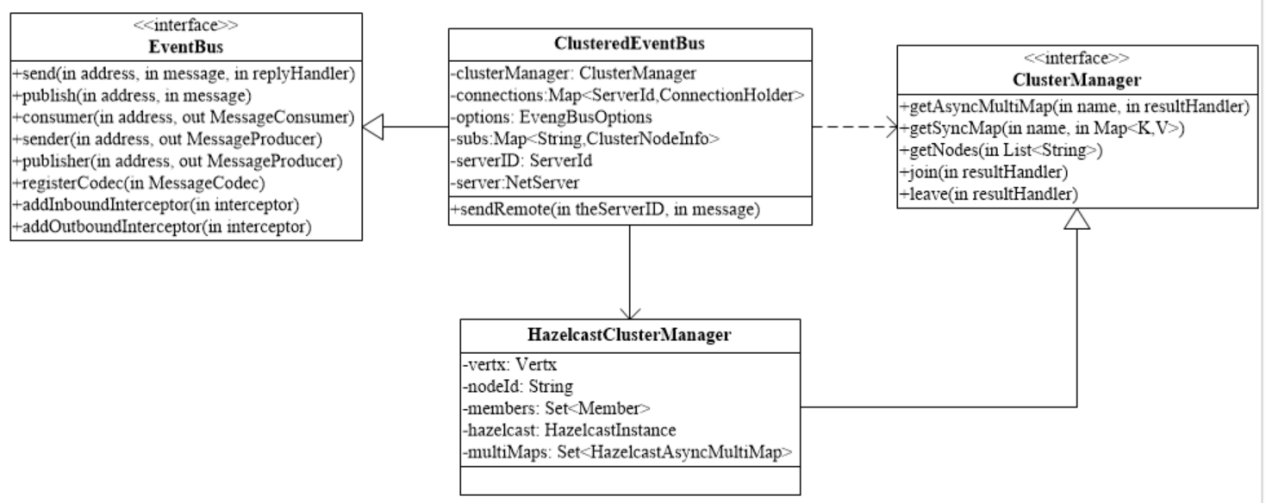
*sendReply(bus.createMessage(true, replyAddress, options.getHeaders(), message, options.getCodecName()), options, replyHandler);*

*}*

*}*

### **Cluster模式ClusteredEventBus**

Cluster EventBus的发送过程与Local Event Bus相同，不同点其使用ClusteredEventBus发送消息，其类图如下：



OutboundDeliveryContext#next中调用clusteredSendReply，其执行如下：

*private <T> void clusteredSendReply(ServerID replyDest,*

*OutboundDeliveryContext<T> sendContext) {*

*MessageImpl message = sendContext.message;*

*String address = message.address();*

*if (!replyDest.equals(serverID)) {*

*sendRemote(replyDest, message);*

*}......*

*}*

sendRemote其执行如下：

*private void sendRemote(ServerID theServerID, MessageImpl message) {*

*ConnectionHolder holder = connections.get(theServerID);*

*if (holder == null) {*

*holder = new ConnectionHolder(this, theServerID, options);*

*ConnectionHolder prevHolder = connections.putIfAbsent(theServerID, holder);*

*if (prevHolder != null) {*

*holder = prevHolder;*

*} else {*

*holder.connect();*

*}*

*}*

*holder.writeMessage((ClusteredMessage) message);*

*}*

在ConnectionHolder中writeMessage的执行如下：

*synchronized void writeMessage(ClusteredMessage message) {*

*if (connected) {*

*Buffer data = message.encodeToWire();*

*socket.write(data);*

*} ....*

*}*

https://vertx.io/docs/vertx-core/java/#streams

https://blog.csdn.net/sdmjhca/article/details/78612792?utm\_source=blogxgwz4

## **异步RPC**

EventBus是Vert.x的核心，使得系统解耦成为可能，但是不能原生支持面向接口的服务调用，因此提供了gRpc模块提供RPC服务组件。

1. 定义proto 文件，service.proto

*syntax = "proto3";*

*option java\_multiple\_files = true;*

*option java\_package = "io.grpc.examples.helloworld";*

*option java\_outer\_classname = "HelloWorldProto";*

*option objc\_class\_prefix = "HLW";*

*package helloworld;*

*// The greeting service definition.*

*service Greeter {*

*// Sends a greeting*

*rpc SayHello (HelloRequest) returns (HelloReply) {}*

*}*

*// The request message containing the user's name.*

*message HelloRequest {*

*string name = 1;*

*}*

*// The response message containing the greetings*

*message HelloReply {*

*string message = 1;*

*}*

1. 定义Server

*public class Server extends AbstractVerticle {*

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

*Runner.runExample(Server.class);*

*}*

*@Override*

*public void start() throws Exception {*

*VertxServer server = VertxServerBuilder.forAddress(vertx, "localhost", 8080).addService(new GreeterGrpc.GreeterVertxImplBase() {*

*@Override*

*public void sayHello(HelloRequest request, Future<HelloReply> future) {*

*System.out.println("Hello " + request.getName());*

*future.complete(HelloReply.newBuilder().setMessage(request.getName()).build());*

*}*

*}).build();*

*server.start(ar -> {*

*if (ar.succeeded()) {*

*System.out.println("gRPC service started");*

*} else {*

*System.out.println("Could not start server " + ar.cause().getMessage());*

*}*

*});*

*}*

*}*

1. 定义客户端

*public class Client extends AbstractVerticle {*

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

*Runner.runExample(Client.class);*

*}*

*@Override*

*public void start() throws Exception {*

*ManagedChannel channel = VertxChannelBuilder*

*.forAddress(vertx, "localhost", 8080)*

*.usePlaintext(true)*

*.build();*

*GreeterGrpc.GreeterVertxStub stub = GreeterGrpc.newVertxStub(channel);*

*HelloRequest request = HelloRequest.newBuilder().setName("Julien").build();*

*stub.sayHello(request, asyncResponse -> {*

*if (asyncResponse.succeeded()) {*

*System.out.println("Succeeded " + asyncResponse.result().getMessage());*

*} else {*

*asyncResponse.cause().printStackTrace();*

*}*

*});*

*}*

*}*

1. 编译执行如下：

*<plugin>*

*<groupId>org.xolstice.maven.plugins</groupId>*

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

*<version>0.5.0</version>*

*<configuration>*

*<protocArtifact>com.google.protobuf:protoc:3.3.0:exe:${os.detected.classifier}*

*</protocArtifact>*

*<pluginId>grpc-java</pluginId>*

*<pluginArtifact>io.vertx:protoc-gen-grpc-java:${vertx.grpc.version}:exe:${os.detected.classifier}*

*</pluginArtifact>*

*</configuration>*

*<executions>*

*<execution>*

*<id>compile</id>*

*<goals>*

*<goal>compile</goal>*

*<goal>compile-custom</goal>*

*</goals>*

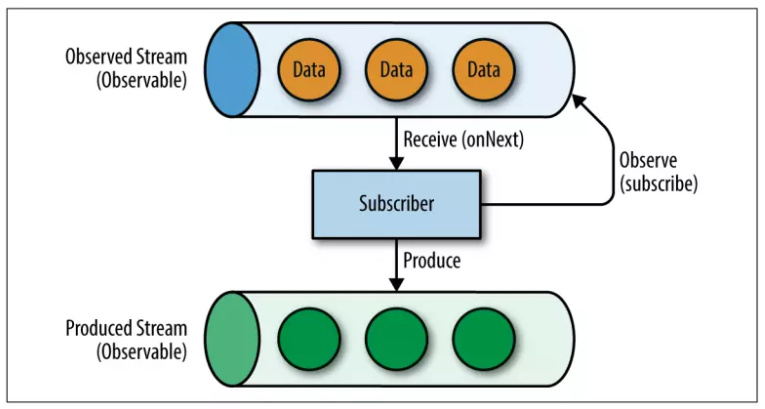
*</execution>*

*</executions>*

*</plugin>*

# **Vert.x Reactive**

Reactive编程是一种开发模型，以数据流和数据传播为驱动，在Reactive编程中刺激信号是数据转移（Streams），如下图：



其中Stream成为observables，消息者对observables进行订阅并且对其值响应。

## **RxJava示例**

1）添加依赖

在Vert.x中提供了RxJava的支持，添加依赖如下：

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-core</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-hazelcast</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-rx-java</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

2） 编写Receiver类

*public class Sender extends AbstractVerticle {*

*@Override*

*public void start() throws Exception {*

*EventBus eb = vertx.eventBus();*

*vertx.setPeriodic(1000,*

*v -> eb.publish("news-feed", "Some news!"));*

*}*

*}*

3）编写Sender类

*public class Receiver extends AbstractVerticle {*

*@Override*

*public void start() throws Exception {*

*EventBus eb = vertx.eventBus();*

*eb.consumer("news-feed")*

*.toObservable()*

*.subscribe( message ->*

*System.out.println("Received news: " + message.body()));*

*System.out.println("Ready!");*

*}*

*}*

4）测试

* 运行Reciver，其输出如下：

*Ready!*

* 运行Sender，在Reciver端输出如下：

*Recycler - -Dio.netty.recycler.ratio: 8*

*Received news: Some news!*

*Received news: Some news!*

## **RxJava源码分析**

RxJava MessageConsumer提供Observable<Message<T>>，

*EventBus eb = vertx.eventBus();*

*MessageConsumer<String> consumer = eb.<String>consumer("the-address");*

*Observable<Message<String>> observable = consumer.toObservable(); //Observable*

*Subscription sub = observable.subscribe(msg -> {*

*// Got message*

*});*

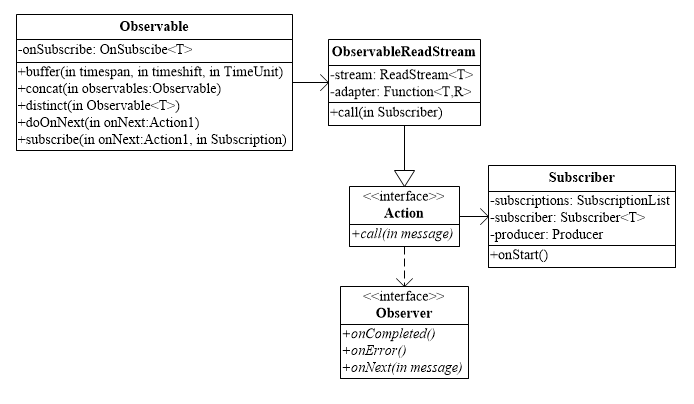
*// Unregisters the stream after 10 seconds*

*vertx.setTimer(10000, id -> {*

*sub.unsubscribe();*

*});*

Observable提供了Message流，subscribe订阅其信息流，并在其流中执行用于自定义的Action1，Observable类图如下所示：



1. Receiver -> EventBus#subscribe，将Action注册到ObservableReadStream，其执行如下：

*static <T> Subscription subscribe(Subscriber<? super T> subscriber,*

*Observable<T> observable) {*

*subscriber.onStart();*

*......*

*try {*

*RxJavaHooks.onObservableStart(observable, observable.onSubscribe).call(subscriber);*

*return RxJavaHooks.onObservableReturn(subscriber);*

*} }*

subscribe后，注册到ReadStream中。以WebSocket为例，其注册如下：

*public S handler(Handler<Buffer> handler) {*

*synchronized (conn) {*

*if (handler != null) {*

*checkClosed();*

*}*

*this.dataHandler = handler;*

*return (S) this;*

*}*

*}*

Subscriber启动后，从ObservableReadStream（EventBus）中读取数据

1. Sender，通过EventBus#pubish生成数据，将其传入到Receiver的Event Bus中，其执行过程已经分析过，这里不再介绍，其执行示例如下：

*vertx.setPeriodic(1000,*

*v -> eb.publish("news-feed", "Some news!"));*

1. Receiver获取数据流，并调用Action（封装成handler）

各数据流入WebSocket/NetSocket/HttpServer等，以WebSocketImpi为例，其通过Handler处理数据，如下所示：

*private void checkNextTick() {*

*if (!paused && queueing && !sentCheck) {*

*sentCheck = true;*

*context.runOnContext(v -> {*

*synchronized (Http1xServerConnection.this) {*

*sentCheck = false;*

*if (!paused) {*

*Object msg = pending.poll();*

*if (pending.isEmpty()) {*

*processMessage(msg);*

*checkNextTick();*

*} }}); }*

*}*

processMessage中调用handler处理message，其如下所示：

*void handleFrame(WebSocketFrameInternal frame) {*

*synchronized (conn) {*

*if (frame.type() != FrameType.CLOSE) {*

*conn.reportBytesRead(frame.length());*

*if (dataHandler != null) {*

*dataHandler.handle(frame.binaryData());*

*}*

*}......*

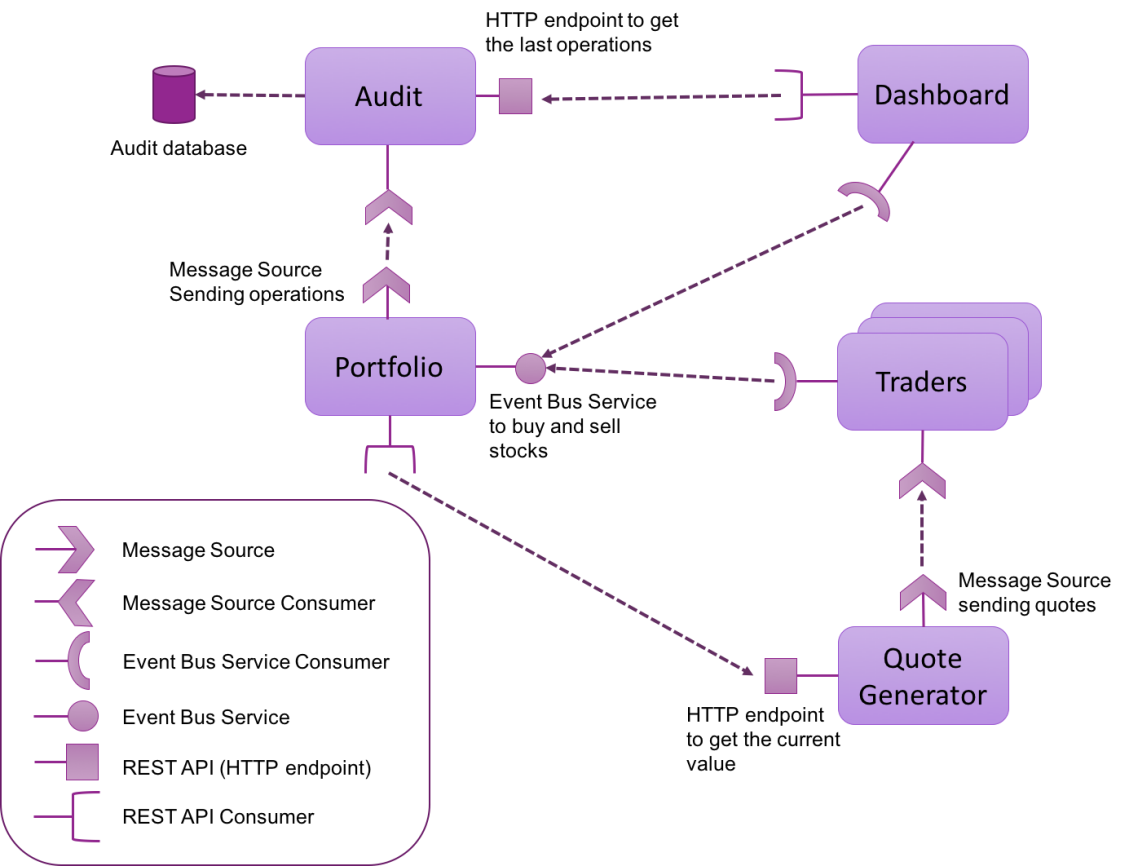
*}*

<https://www.jianshu.com/p/b469f1c7071d>

<https://blog.csdn.net/king866/article/details/55215337>

# **Vert.x MicroService**

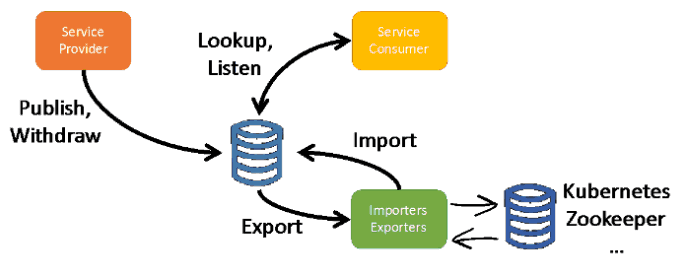
微服务是目前软件架构的首先方案，Vert.x是建立微服务的工具，其提供异步的通信方式，类似于actor模型的可部署和并发模型。其中Verticles是代码块，能够部署和运行在Vert.x中，Vert.x应用程序通常由许多Verticles实例运行在Vert.x中，实例间通过Event Bus发送消息。示例如下：



[https://github.com/cescoffier/vertx-microservices-workshop](https://github.com/cescoffier/vertx-microservices-workshop" \t "https://www.cnblogs.com/luxiaoxun/p/_blank)

## **服务发现**

Vert.x提供了服务发现机制，让服务提供者发布服务，让消费者找到正确的服务，其支持的服务发现机制如Zookeeper、Consul及Kubenets等



服务发现机制的代码中，增加下面的依赖：

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-service-discovery</artifactId>*

*<version>3.5.3</version>*

*</dependency>*

发布一条记录到Vert.x服务发现中，其示例如下：

*ServiceDiscovery discovery = ServiceDiscovery.create(vertx,*

*new ServiceDiscoveryOptions()*

*.setAnnounceAddress("service-announce")*

*.setName("my-name"));*

*//create a new custom record*

*Record record1 = new Record()*

*.setType("eventbus-service-proxy")*

*.setLocation(new JsonObject().put("endpoint", "the-service-address"))*

*.setName("my-service")*

*.setMetadata(new JsonObject().put("some-label", "some-value"));*

*//publish "my-service" service*

*discovery.publish(record1, ar -> {*

*if (ar.succeeded()) {*

*System.out.println("\"" + record1.getName() + "\" successfully published!");*

*Record publishedRecord = ar.result();*

*} else {*

*// publication failed*

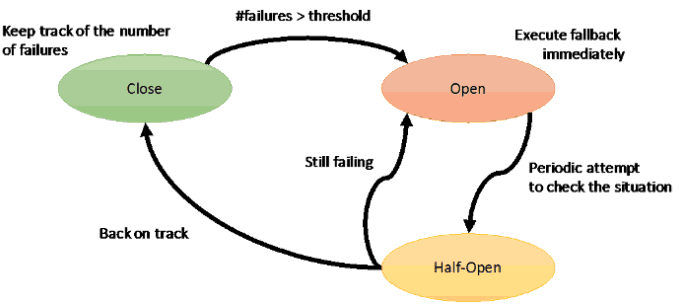
*}*

*});*

Vert.x将这些记录存储在分布式存储中分享给Vert.x集群中的所有成员，其支持的配置包括分布式map、Apache Zookeeper等。

## **断路器**

断路器是微服务的弹性支持的核心模块，其优点是恢复功能，断路器的状态机如下所示：



断路器监视操作，并跟踪给操作的故障数，当达到阈值则通过断路器阻止该操作。其依赖如下：

<*dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-circuit-breaker</artifactId>*

*<version>${project.version}</version>*

*</dependency>*

使用示例如下：

*CircuitBreakerOptions options = new CircuitBreakerOptions()*

*.setMaxFailures(5)*

*.setTimeout(5000)*

*.setFallbackOnFailure(true);*

*CircuitBreaker breaker =*

*CircuitBreaker.create("my-circuit-breaker", vertx, options)*

*.openHandler(v -> {*

*System.out.println("Circuit opened");*

*}).closeHandler(v -> {*

*System.out.println("Circuit closed");*

*});*

*Future<String> result = breaker.executeWithFallback(future -> {*

*vertx.createHttpClient().getNow(8080, "localhost", "/", response -> {*

*if (response.statusCode() != 200) {*

*future.fail("HTTP error");*

*} else {*

*response.exceptionHandler(future::fail).bodyHandler(buffer -> {*

*future.complete(buffer.toString());*

*});*

*}*

*});*

*}, v -> {*

*// Executed when the circuit is opened*

*return "Hello (fallback)";*

*});*

*result.setHandler(ar -> {*

*// Do something with the result*

*System.out.println("Result: " + ar.result());*

*});*

Netflix Hystric支持Dashboard，用于显示熔断器的当前的状态，Vert.x熔断器可以发布其指标，以供Hystrix仪表板使用。

## **配置**

Vert.x Config提供配置Vert.x的应用，包含：

* 支持多种配置语法（JSON、properties、YAML（扩展）、HOCON（扩展）等）
* 支持多种存储方式（文件、目录、HTTP、Git（扩展）、Redis（扩展）、系统属性、环境变量等）
* 支持自定义配置处理顺序和配置重载(overloading)
* 支持运行时重新配置(runtime reconfiguration)

在该组件中基于下面几个概念：

* Config Retriever，由Vert.x应用实例化以及使用，其配置了一系列的配置仓库
* Configuration Store，定义了配置项的数据源和语法，默认JSON

其使用示例如下，在依赖中增加配置如下：

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-config</artifactId>*

*<version>3.4.1</version>*

*</dependency>*

第一步做的是实例化ConfigRetriever，示例如下：

*ConfigRetriever retriever = ConfigRetriever.create(vertx);*

通过下列自定义配置：

*ConfigStoreOptions httpStore = new ConfigStoreOptions()*

*.setType("http")*

*.setConfig(new JsonObject()*

*.put("host", "localhost").put("port", 8080).put("path", "/conf"));*

*ConfigStoreOptions fileStore = new ConfigStoreOptions()*

*.setType("file")*

*.setConfig(new JsonObject().put("path", "my-config.json"));*

*ConfigStoreOptions sysPropsStore = new ConfigStoreOptions().setType("sys");*

*ConfigRetrieverOptions options = new ConfigRetrieverOptions()*

*.addStore(httpStore).addStore(fileStore).addStore(sysPropsStore);*

*ConfigRetriever retriever = ConfigRetriever.create(vertx, options);*

开源参考：

https://vertxchina.github.io/vertx-translation-chinese/core/Core.html

https://github.com/vert-x3/vertx-examples

https://github.com/eclipse-vertx/vert.x

<https://vertx.io/>

https://github.com/vert-x3

简介：

https://www.jianshu.com/p/3c9d69df44dc

https://www.cnblogs.com/luxiaoxun/p/7693640.html

<http://www.eclipse.org/community/eclipse_newsletter/2016/october/article4.php>

Context:

https://blog.csdn.net/ifrozen/article/details/52650554

https://blog.csdn.net/JflyDragon/article/details/52522844

Vert.x，使用Hazelcast来创建和管理集群：

https://my.oschina.net/chkui/blog/678347

<https://vertx.io/docs/vertx-hazelcast/java/>

Apache Shiro:

<https://blog.csdn.net/xtiawxf/article/details/52606653>

https://bitbucket.org/tomask79/spring-leader-hazelcast/overview