vert.x框架：

1. 创建一个基础的Http服务器：

创建一个maven项目，导入vert.x的core依赖

直接在主程序中创建该服务器：（项目使用的jre必须为1.8版本）

public static void main(String[] args) {

// 创建一个vertx对象

Vertx vertx = Vertx.vertx();

// 通过vertx对象创建一个Httpserver

HttpServer Server = vertx.createHttpServer();

// 配置该服务器的request处理器；将server的requestHandler方法的回调值HttpServerRequest最为参数，传递到后面的lambda表达式中

通过该服务器的request对象来绑定一个response对象（设置其内容格式和编码格式）

Server.requestHandler(re ->{

re.response().putHeader("Content-type","text/html;charset=utf-8").end("hello 哈哈哈");

}).listen(8881);

}

\*\*\*\*\*创建Vertx对象时，可以修改其默认值

Vertx vertx = Vertx.vertx(new VertxOptions().setWorkerPoolSize(40));

如高可用性，工作池大小等

1. 代码改进：创建服务器的代码需要写在main中来运行，这样不太好

Vert.x推荐将每一个应用都创建一个Verticle；通过Vertx核心接口来部署Verticle模块；

然后每个Verticle模块都可以通过EventBus来进行相互调用

Verticle是在Vert.x应用程序部署的基本单位。

》》》创建一个Verticle类（需要实现Verticle接口），一般可以通过继承AbstractVerticle类（verticle接口的实现类）来直接实现

继承了AbstractVerticle这个类，在这个类提供一个protected类型的Vertx变量，通过该变量在类中创建HttpServer对象

》》》然后重写AbstractVerticle类的start（） 部署Verticle时调用，用于编写该Verticle模块的应用代码

---也可以选择性重写stop方法， 在取消部署Verticle时调用

当Vert.x部署Verticle时，它将调用start方法，当方法完成时，将认为Verticle已启动。

您也可以选择覆盖stop方法。当取消部署Verticle时，Vert.x会调用此方法，当方法完成时，Verticle将被视为已停止。

Eclipse中关闭创建的服务器，使用stop图标，防止端口被占用

public class MyVerticle extends AbstractVerticle{

public void start(){

// 创建一个HttpServer对象（通过vertx实例对象）

HttpServer server =vertx.createHttpServer();

// 设置该Http服务器的响应头和响应数据

server.requestHandler( request -> {

// 通过server的request对象获的response

HttpServerResponse response = request.response();

// 设置响应头

response.putHeader("Content-type","text/html;charset=utf-8");

// 响应数据

response.end("Helo World");

}).listen(8844);

}

}

public class MyServer {

public static void main(String[] args) {

// 调用创建的Verticle类

MyVerticle myVerticle = new MyVerticle();

// 创建一个Vertx对象

Vertx vertx = Vertx.vertx();

// 通过vertx对象的deployVerticle方法来部署Verticle，此时就会自动调用Verticle模块中的start方法

vertx.deployVerticle(myVerticle);

}

}

Vertx处理程序是通过事件来触发的，比如创建一个Http服务器，当有一个request请求传递事件发生时，执行对应的代码

1. Vert.x框架的Web开发：

在Vert.x 创建HTTP服务 中我们已经创建了一个简单的HttpServer，但这个HttpServer比较低级，对于请求参数解析、Session等常用功能都需要我们通过编码实现，非常不方便。

Vert.x提供了Web开发组件vertx-web，提供了一堆Web开发中常用的功能。比如参数封装，路由，国际化，认证和授权，session和cookie以及模板等，可以非常方便的进行Vert.xWeb开发。

我们一般使用Vertxweb组件的路由功能：

路由的作用：

简单说就是把用户请求交给合适的处理器处理的组件

类似于springMVC中的前端控制器截获请求匹配对应的controller（控制器），来分析处理请求，根据不同的url来执行不同的HttpServer的requestHandler（处理器）



创建Http服务器，通过路由来处理请求:

public void start(){

// 创建HttpServer对象

HttpServer createHttpServer = vertx.createHttpServer();

// 创建路由对象(通过vertx-web包提供的Router接口的方法，以vertx对象最为参数创建其接口的实例)

Router router = Router.router(vertx);

// 将服务器请求提交给路由的处理器处理（将requestHandler方法的回调值HttpServerRequest作为参数，传递路由对象的accept方法中；这样请求就可以交个路由的对应处理器处理）

createHttpServer.requestHandler(router::accept);

// 监听/index地址(通过路由对象来设置其用于处理请求的处理器){当请求地址为/index时，通过匹配交个此时的handler处理}

router.route("/index").handler(request ->{

request.response().end("index success");

});

// 设置该服务器的监听器端口

createHttpServer.listen(8877);

}

}

》》》》对于路由匹配url来选择不同的处理器，有不同的匹配方式：

1. router.route( “url”) 通过route方法直接设置匹配的url（默认使用GET的提交方式）
2. router. Post(“url”)

除了post方法外，还有get、put等方法

根据不同的方法，来匹配对应HTTP请求方式

1. router.route(HttpMethod.GET ，url)

route()方法还有另一种重载，参数有两个，前一个为Http请求方式（Method），后一个为匹配请求的url

》》》》指定路由的匹配顺序：

Vert.x进行路由匹配时，默认从上到下匹配，匹配到时，不在继续进行

但有时的确存在这种问题，如：

/index/\*和/index/main 此时，请求为/index/main,我们只需要匹配后者，但前者也会被匹配，导致没有实现正确的处理，因此对于这个问题有两种解决方式：

1. 我们对某些优先级需求高一点的存在相同的监听url处理器设置代码放在上面，但解决该问题的程度有限，不能完全
2. 制定order，对可能存在相同匹配的处理器添加order优先级

router.route(HttpMethod.GET,"/index/\*").order(-3)

order中的值越小，匹配优先级越高，可以为负数

1. 也可以匹配所有规则相同的处理器

只需要在处理器设置代码最后添加一条代码：

request.next(); // 调下一个匹配规则

》》》》》参数传递

1. 通过get请求在url后传递参数

如：localhost：8080/index?name=袁欢

（中文会乱码

String suppliername = request.getParameter("suppliername");

suppliername= new String(suppliername.getBytes("iso8859-1"),"UTF-8");）

然后通过request.request().getParam(“name”) 通过路由作为中间件时，初始的request对象为路由的，需要通过request（）方法获的Httpservelt的request对象

1. 通过获取路径参数（通过RESTful（一种软件系统架构风格）处理url）

Url=/index/yuanhuan/1234

router.route(HttpMethod.GET, "/index/:user/:pass")

1. 获的POST中提交的数据

// 路由默认创建（即路由被实例化后就创建）一个Body体处理器，用于处理request请求中的body体

router.route().handler(BodyHandler.create());

之后再创建处理器，获取request请求中的body体数据

router.route(HttpMethod.POST,"/post").handler(request ->{

// 获的POST提交的数据（以字符串类型获的），通过body处理器处理后的数据

String String = request.getBodyAsString();

})

》》》》》实际应用中，会有大量的处理器的逻辑代码，这样会导致代码结构不清晰

因此可以将监听和具体的请求处理逻辑分开：

1. 创建一个类，并实现Handler接口，类属于vertx.core包，且该接口类型为RoutingContext类型（路由上下文）
2. 重写Handler接口的handle方法，其代码则为该处理器的请求处理逻辑代码

public class MyHandeler implements Handler<RoutingContext>{

@Override

public void handle(RoutingContext event) {

// TODO Auto-generated method stub

}

}

1. 在Verticle类中添加对url的处理器监听

Router.route(“url”).handler(new myhandle()）调用自定义的handler类来实例化路由的handler接口

》》》》》vertx框架提供多个默认的处理器，如：BodyHandler、SessionHandler、CookieHandler

》》》》vertx框架提供二级路由功能，用于编辑路由处理规则时，太过于杂乱

可以将类型相同的处理器监听，编辑在同一个二级路由中，最后在挂载在唯一的主路由里;

\*\*\*\*\*这样一种模块的所有处理器，就放在一个二级路由中；通过多个二级路由，完成所有处理器的设置，然后全部通过一个主路由来挂载。

1. 在自定义的Verticles类中，创建服务器
2. 创建二级路由，并编辑其处理器监听
3. 创建主路由，并挂载创建好的二级路由
4. 将服务器的请求提交个主路由处理，并设置服务器端口

public class Verticles1 extends AbstractVerticle{

public void start(){

// 创建服务器

HttpServer server = vertx.createHttpServer();

// 创建二级路由

Router router = Router.router(vertx);

// 设置路由的监听地址

router.route("/hhh").handler(new Handeler1());

router.route("/he").handler(new Handler2());

// 创建一个主路由

Router Mainrouter = Router.router(vertx);

// 主路由挂载二级路由

Mainrouter.mountSubRouter("/main",router);

// 将该服务器的请求交给主路由处理

server.requestHandler(Mainrouter::accept);

// 设置该服务器的监听端口

server.listen(8888);

}

}

二级路由其实就类似于spirngMVC中的controller类的@RequestMapping注解，（或strut2配置文件中的namespace）用于添加处理器匹配的url段，从而增加匹配url的区分度，将不同类型handler区分开来，从而在同时开发时不会存在匹配的url相同的情况

1. Vertx实现TCP服务

1、创建TCP服务器，处理连接请求

创建一个Verticle模块，在start（）方法中编写执行代码

public class Verticle\_tcp extends AbstractVerticle{

@Override

public void start() throws Exception {

// 创建TCP服务器

NetServer server = vertx.createNetServer();

// 处理连接请求

server.connectHandler(socket ->{

通过server的NetSocket对象处理请求（SocketBuffer中的数据）

socket.handler(buffer ->{

// 接收客户端传递来的请求数据

System.out.println(buffer.toString());

// 通过协议响应给客户端

socket.write(Buffer.buffer("hahhaha"));

});

// 客户端接收响应后，连接关闭

socket.closeHandler(smg ->{

System.out.println("连接关闭");

});

});

// 设置该TCP服务器的监听端口

server.listen(8808, res->{

System.out.println("服务器上线");

});

1. 创建一个TCP客户端

创建一个Verticle模块，在start方法中编写执行代码

public class Verticle\_tcpClient extends AbstractVerticle{

@Override

public void start() throws Exception {

// 创建TCP客户端

NetClient netClient = vertx.createNetClient();

// 连接服务器（设置连接服务器的端口，IP；连接时执行下面代码）

netClient.connect(8808,"localhost", conn->{

if (conn.succeeded()) {

// 如果连接成功,通过连接对象创建socket对象

NetSocket result = conn.result();

// 在socket的Buffer中添加要传输的数据

result.write(Buffer.buffer("hello"));

// 处理服务器传递过来的响应数据(存放在socket buffer中)

result.handler(buffer ->{

System.out.println(buffer.toString());

});

}else{

// 如果连接失败

System.out.println("服务器连接异常");

}

});

}

}

1. Vertx连接mysql数据库

导入Vertx-core、vertx-jdbc、mysql驱动包

<dependency>

<groupId>io.vertx</groupId>

<artifactId>vertx-jdbc-client</artifactId>

<version>3.6.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.13</version>

</dependency>

1. 创建一个工具类，用于创建JDBCClient对象（JDBC客户端），来处理数据库

public class JDBCUtils {

// 创建jdbc客户端对象（用于操作数据库的）

private JDBCClient jdbcClient;

// 有参构造方法(通过vertx对象，获的JDBCClient对象)

public JDBCUtils(Vertx vertx){

// 创建一个json对象，存放JDBC四要素

JsonObject jsonObject=new JsonObject();

// 由于mysql数据库时区问题，需要在url后修改数据库时区?serverTimezone=GMT%2B8

jsonObject.put("url", "jdbc:mysql://localhost:3306/hibernatedemo?serverTimezone=GMT%2B8");

jsonObject.put("driver\_class", "com.mysql.cj.jdbc.Driver");

jsonObject.put("user", "root");

jsonObject.put("password", "");

// 通过四要素来创建JDBC客户端对象

jdbcClient=JDBCClient.createShared(vertx, jsonObject);

}

// 提供JDBC客户端对象属性的get、set方法

public JDBCClient getJdbcClient() {

return jdbcClient;

}

public void setJdbcClient(JDBCClient jdbcClient) {

this.jdbcClient = jdbcClient;

}

}

1. 创建一个Verticle，编写处理数据库的逻辑代码

public class JdbcTextVerticle extends AbstractVerticle{

@Override

public void start() throws Exception {

// 通过JDBCUtils工具类，获的JDBC客户端(用于查询数据库)

JDBCClient jdbcClient = new JDBCUtils(vertx).getJdbcClient();

// 编写sql语句(通过？标识符进行参数传递)

String sql="select \* from account where id=?";

// 创建一个Json数组，作为sql语句的参数容器

JsonArray sqlParams=new JsonArray();

// 将参数数据放入数组

sqlParams.add(2);

// 通过JDBC客户端执行sql语句

// queryWithParams 需要sql语句、sql参数、还有返回结果做为参数的回调函数

jdbcClient.queryWithParams(sql, sqlParams, res->{

if(res.succeeded()){

// 如果查询有结果

// 获得的res<ResultSet>是被vertx封装好的，需要通过result（）获的封装数据、

ResultSet result = res.result();

// 转换数据类型（将结果集合类型转换为List<JsonObject>json类型的list集合）

List<JsonObject> rows = result.getRows();

// 输出该list集合数据（可循环遍历，获的每一个json对象数据）

System.out.println(rows);

}else {

// 如果查询没有结果

System.out.println("查询数据出错");

}

});

}

}

1. 通过在main方法中，创建vertx实例，来部署Verticle模块

public static void main(String[] args) {

Vertx vertx = Vertx.vertx();

vertx.deployVerticle(new JdbcTextVerticle());

}

数据库连接优化：

类似于orm框架数据库的处理，对于数据库连接设置，都是通过配置文件或使用数据库连接池，从而支持多数据源，方便修改添加其他数据库配置

1. 使用配置文件：

创建一个配置文件，通过读取该配置文件来创建一个存放JDBC四要素的JSON数组

1)创建file类型文件；文件类型为db.json,

{

"default":{

"url":"jdbc:mysql://localhost:3306/hibernatedemo?serverTimezone=GMT%2B8",

"driver\_class":"com.mysql.jdbc.Driver",

"user":"root",

"password":""

},

"prod":{

"url":"jdbc:mysql://localhost:3306/hibernatedemo?serverTimezone=GMT%2B8",

"driver\_class":"com.mysql.jdbc.Driver",

"user":"root",

"password":""

}

}

2)修改JDBCUtils工具类（用于创建JDBCClient对象）

// 定义一个静态的json对象，最为jdbc配置文件(通过下面的静态代码，在程序被编译时，就读取创建好对应的JSON对象)

private static JsonObject config;

// 编写一段静态代码

static{

// 创建一个100KB的字节数组

byte[] buff =new byte [102400];

try {

// 获的项目绝对路径

String property = System.getProperty("user.dir");

// 获的项目中json文件的绝对路径

String realPath=property+"/src/main/resources/db.json";

// 创建一个输入流，来读取配置文件数据

FileInputStream fis = new FileInputStream(new File(realPath));

// 将输入流数据写入buff数组中（commons-io插件包中的IOUtils写入）

int i=IOUtils.read(fis, buff);

// 通过读取的数据创建一个json对象

config=new JsonObject(new String(buff, 0,i));

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

System.out.println("读取配置文件失败");

}

}

// 提供一个有参构造方法，来创建JDBC客户端对象

public JDBCUtils(Vertx vertx,String dsName){

// 在配置文件的JSON对象中，通过KEY来选择对应的连接数据，用JSON对象保存

JsonObject dbconfig=config.getJsonObject(dsName);

if (dbconfig==null) {

// 如果通过KEY找不到对应的json数据

System.out.println("找不到对应的数据源");

}else{

// 创建JDBC客户端对象

this.jdbcClient = JDBCClient.createShared(vertx, dbconfig);

}

}

2、使用数据库连接池：

通过vertx创建的HTTP服务器，调用静态资源（css、js、HTML）：  
1、首先建立一个HTTP服务器，并将请求处理交给路由

2、设置路由的处理

Vertx认证和授权：

使用Vert.x认证和授权，大概是经历三个阶段

1. 自己实现AuthProvider和User接口实现一个简单的认证和授权。

2. 使用Vert.x提供的授权方式，如JDBC

3. 在Web中使用认证和授权来管理访问权限

一、1. 自定义授权实现

实现AuthProvider和User两个接口，重写两接口的认证方法

需要导入Vertx-auth依赖包

<dependency>

<groupId>io.vertx</groupId>

<artifactId>vertx-auth-common</artifactId>

<version>3.6.2</version>

</dependency>

Vertx事件总线（EventBus）与远程服务调用

Event Bus 是Vert.x的神经系统，负责Vert.x各Verticle模块之间的消息的传送。

EventBus提供发布订阅功能和点对点的消息服务,远程服务调用就是通过EventBus的点对点消息服务实现的。

每条消息在EventBus上都有一个地址，发布者向这个地址发送消息，接收者从这个地址接收消息。

EventBus是使用TCP协议进行通信，因此，在任何的应用中，只要能够创建TCP连接，都可以通过EventBus连接到Vert.x实例。

1、通过Vert.x获得EventBus实例

EventBus eb=vertx.eventBus（）

// 通过vertx对象获得EventBus（事件总线）

EventBus eb = vertx.eventBus();

// 发布消息（所有人都可以通过address获得该消息）

eb.publish(address, message)

// 通过点对点的方式发布消息（其他人接收不到）

eb.publish(address, message, options)

// 订阅消息(获得消息，并进行处理)

eb.consumer(address, handler)