# 1.mybatis

## 1.1.mybatisPlus

### 1.1.1 mybatisPlus的配置文件

|  |
| --- |
| # Mybatis-Plus 配置  mybatis-plus:  # mapper-locations: classpath:/mapper/\*Mapper.xml  #实体扫描，多个package用逗号或者分号分隔  typeAliasesPackage: com.tensquare.article.pojo  configuration:  log-impl: org.apache.ibatis.logging.stdout.StdOutImpl #打印sql语句  global-config:  id-type: 1 #0:数据库ID自增 1:用户输入id  db-column-underline: false  refresh-mapper: true  configuration:  map-underscore-to-camel-case: true #实体类属性驼峰命名规则  cache-enabled: true #配置的缓存的全局开关  lazyLoadingEnabled: true #延时加载的开关  multipleResultSetsEnabled: true #是否允许单一语句返回多结果集（需要兼容驱动） |

**提示**：

|  |
| --- |
| Mybatis Plus 提供的EntityWrapper对象封装查询条件  mybatisPlus就是设置好pojo类与表之间的映射关系--------》和通用mapper的区别是，通用mapper需要在pojo类中与数据库表建立一种全局映射关系  而mybatisPlus是在application.yml中配置一些定义  同时mybatis-plus还提供了一些较为常用的用法---》分页查询和条件查询等 |

### 1.1.2mybatisPlus的mapper接口

|  |
| --- |
| @Service  public class ArticleService {  } |

这里可以不用写接口，而直接用实体类。只是代理的方式不一样，有接口使用jdkproxy 没接口使用cglib

# 2.全局异常处理类

## 2.1.创建自定义异常

|  |
| --- |
| public class MyException extends RuntimeException {  public MyException(String message) {  super(message);  }  } |

## 2.2.异常捕获

|  |
| --- |
| @RestControllerAdvice  public class MyExceptionAdvice {  @ExceptionHandler(Exception.class)  public Result handException(Exception e){  e.printStackTrace();  return new Result(false, StatusCode.ERROR,"发生异常了，请联系管理员");  }    //捕获自定义异常  @ExceptionHandler(MyException.class)  public Result handMyException(Exception e){  e.printStackTrace();  return new Result(false, StatusCode.ERROR,e.getMessage());  }  } |

## 2.3.抛出自定义异常

|  |
| --- |
| //抛出自定义的异常  if (StringUtils.isEmpty(map.get("columnid"))) {  throw new MyException("这是我抛出的异常");  } |

# 3.跨域

## 3.1网关

### 3.1.1 GateWay网关微服务

### 3.1.2 Zuul网关微服务

1.添加依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  </dependency>  </dependencies> |

2.配置文件

|  |
| --- |
| server:  port: 9013  spring:  application:  name: tensquare-encrypt  zuul:  routes:  tensquare-article: #文章微服务  path: /article/\*\* #配置请求URL的请求规则  serviceId: tensquare-article #指定Eureka注册中心中的服务id（指定要拦截那个微服务）  strip-prefix: true  sentiviteHeaders: #将指定路由的敏感头设置为空  customSensitiveHeaders: true #对指定路由开启自定义敏感头  eureka:  client:  service-url:  defaultZone: http://127.0.0.1:6868/eureka/  instance:  prefer-ip-address: true |

3.启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableZuulProxy  public class EncryptApplication {  public static void main(String[] args) {  SpringApplication.run(EncryptApplication.class);  }  } |

## 3.2 @CrossOrigin

|  |
| --- |
| @RestController @RequestMapping***(*"/article"*)*** @CrossOrigin *//跨域* **public class** ArticleController ***{  …………***  } |

# 4.MongoDB

## 4.1 spring整合操作数据库

4.1.1 依赖

|  |
| --- |
| ***<*dependency*>  <*groupId*>***org.springframework.boot***</*groupId*>  <*artifactId*>***spring-boot-starter-data-mongodb***</*artifactId*> </*dependency*>*** |

4.1.2：配置文件

|  |
| --- |
| **data:  mongodb:  database:** commentdb *#数据库的名称* **host:** 192.168.211.131  **port:** 27017 |

4.1.3 操作数据库方式  **两种方式（spring-data整合以及原生的模板---很多第三方中间件都是这样，redisTemplate以及jdbcTemplate….）**

|  |
| --- |
| 方式一： spring封装  //持久层  **public interface** CommentRepository **extends** MongoRepository***<***Comment,String***> {*** *// springdata中通过方法命名方式查询  // findBy+属性名(首字母要大写)* List***<***Comment***>*** findByArticleId***(***String articleId***)***;//这个是自定义方法  ***}***  //service层  **commentRepository**.findAll***()***;  方法二：使用模板类  *//原生的查询类 有更新器 如果需要使用mongo的原生命令，就得用mongoTemplate*  @Autowired **private** MongoTemplate **mongoTemplate**;  **public void** thumbup***(***String id***) {*** Query query = **new** Query***()***;  query.addCriteria***(***Criteria.*where****(*"\_id"*)***.is***(***id***))***;   Update update = **new** Update***()***;  update.inc***(*"thumbup"**,1***)***;  **mongoTemplate**.updateFirst***(***query,update,**"comment"*)***;  ***}*** |

# 5．拦截过滤器

你继承实现了那种类型的过滤器，便具有那种作用，通过order可以设定此过滤器的执行顺序，在拦截之后，你要设定拦截之后的动作

过滤有序则设置order的大小，order数值越小越优先

|  |
| --- |
| @Component **public class** RSARequestFilter **extends** ZuulFilter ***{***  /\*\*  \* 定义过滤器的类型  \* @return  \*/  @Override  public String filterType() {  // 前置过滤，在转发微服前执行  return "pre";  }  /\*\*  \* 过滤器执行的顺序  \* 数值越小，越优先执行  \* @return  \*/  @Override  public int filterOrder() {  return 0;  }  /\*\*  \* 是否过滤（启用）  \* 返回true时，才会调用run方法  \* @return  \*/  @Override  public boolean shouldFilter() {  return true;  }  /\*\*  \* 过滤的处理方法  \* @return  \* @throws ZuulException@Override  **public** Object run***()* throws** ZuulException ***{***  ***}***  ***}*** |

# 6.微服务-SpringCloud

## 6.1引入SpringCloud依赖

|  |
| --- |
| 一般在父工程中引入  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Greenwich.SR1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

## 6.2引入注册中心

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  </dependency>  </dependencies> |

## 6.3添加注册中心配置

|  |
| --- |
| server:  port: 6868  eureka:  client:  register-with-eureka: false #是否将自己注册到eureka中  fetch-registry: false #是否从eureka中获取信息  service-url:  defaultZone: http://127.0.0.1:${server.port}/eureka/ |

## 6.4.编写启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaServer  public class EurekaApplication {  public static void main(String[] args) {  SpringApplication.run(EurekaApplication.class, args);  }  } |

===========以上是编写注册中心微服务

===========以下是微服务注册到注册中心

## 6.5添加Eureka依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  </dependency> |

## 6.6修改配置文件，使用Eureka

1.defaultZone: <http://127.0.0.1:6868/eureka/>:

未配置eureka.client.serviceUrl.defaultZone值，EurekaClientConfigBean设置的默认值http://localhost:8761/eureka/不会被覆盖

2.prefer-ip-address: true #以IP地址注册到服务中心，相互注册使用IP地址

单节点部署为分布式Eureka集群，设置当前机器hostname为多个，不同节点eureka.instance.hostname设置为不同值，同时prefer-ip-address禁止设置为true，否则DS Replicas列表中会出现不应该出现的节点。

|  |
| --- |
| eureka:  client:  service-url:  defaultZone: <http://127.0.0.1:6868/eureka/>  instance:  prefer-ip-address: true #以IP地址注册到服务中心，相互注册使用IP地址 |

## 6.6在微服务中添加@EnableEurekaClient依赖

|  |
| --- |
| @SpringBootApplication  //配置Mapper包扫描  @MapperScan("com.tensquare.article.dao")  @EnableEurekaClient  //(注册中心微服务为@EnableEurekaServer,使用的为@EnableEurekaClient)  public class ArticleApplication {  public static void main(String[] args) {  SpringApplication.run(ArticleApplication.class, args);  }  } |

# 7.Spring-Test测试类

7.1引入依赖

|  |
| --- |
| ***<*dependency*>  <*groupId*>***org.springframework.boot***</*groupId*>  <*artifactId*>***spring-boot-starter-test***</*artifactId*>  <*scope*>***test***</*scope*> </*dependency*>*** |

7.2使用

|  |
| --- |
| @RunWith***(***SpringJUnit4ClassRunner.**class*)*** @SpringBootTest***(***classes= EncryptApplication.**class*)//启动类* public class** EncryptTest ***{***  ………………………  } |

# 8.redis的使用

## 1.点赞功能

|  |
| --- |
| *//模拟当前用户* String userId = **"4"**;  *//控制重复点赞—看redis中此用户是否有值OK，有的话这说明点赞了* Object value = **redisTemplate**.opsForValue***()***.get***(*"article\_"** + userId + **"\_"** + articleId***)***; **if *(***value != **null*) {* return new** Result***(*false**, StatusCode.***REPERROR***, **"不能重复点赞"*)***; ***}***  ***//没有值，将点赞数+1，修改用户的redis集合状态* articleService**.thumbup***(***articleId***)***; **redisTemplate**.opsForValue***()***.set***(*"article\_"** + userId + **"\_"** + articleId,**"OK"*)***; **return new** Result***(*true**, StatusCode.***OK***, **"点赞成功"*)***; |

# 9.WenSocket

全双工通信

# 10.多线程

## 10.1传统的IO

10.1.1服务端

|  |
| --- |
| 一个线程处理一个连接  public class IOServer {  public static void main(String[] args) throws Exception {  ServerSocket serverSocket = new ServerSocket(8000);  while (true) {  // (1) 阻塞方法获取新的连接  Socket socket = serverSocket.accept();  new Thread() {  @Override  public void run() {  String name = Thread.currentThread().getName();  try {  // (2) 每一个新的连接都创建一个线程，负责读取数据  byte[] data = new byte[1024];  InputStream inputStream = socket.getInputStream();  while (true) {  int len;  // (3) 按字节流方式读取数据  while ((len = inputStream.read(data)) != -1) {  System.out.println("线程" + name + ":" + new String(data, 0, len));  }  }  } catch (Exception e) {  }  }  }.start();  }  }  } |

10.1.2客户端

|  |
| --- |
| public class MyClient {  public static void main(String[] args) {  //测试使用不同的线程数进行访问  for (int i = 0; i < 5; i++) {  new ClientDemo().start();  }  }  static class ClientDemo extends Thread {  @Override  public void run() {  try {  Socket socket = new Socket("127.0.0.1", 8000);  while (true) {  socket.getOutputStream().write(("测试数据").getBytes());  socket.getOutputStream().flush();  Thread.sleep(2000);  }  } catch (Exception e) {  }  }  }  } |

## 10.2 NIO

|  |
| --- |
| 一个线程处理多个连接  public class NIOServer {  public static void main(String[] args) throws IOException {  // 负责轮询是否有新的连接  Selector serverSelector = Selector.open();  // 负责轮询处理连接中的数据  Selector clientSelector = Selector.open();  new Thread() {  @Override  public void run() {  try {  // 对应IO编程中服务端启动  ServerSocketChannel listenerChannel = ServerSocketChannel.open();  listenerChannel.socket().bind(new InetSocketAddress(8000));  listenerChannel.configureBlocking(false);  // OP\_ACCEPT表示服务器监听到了客户连接，服务器可以接收这个连接了  listenerChannel.register(serverSelector, SelectionKey.OP\_ACCEPT);  while (true) {  // 监测是否有新的连接，这里的1指的是阻塞的时间为1ms  if (serverSelector.select(1) > 0) {  Set<SelectionKey> set = serverSelector.selectedKeys();  Iterator<SelectionKey> keyIterator = set.iterator();  while (keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();  if (key.isAcceptable()) {  try {  // (1) 每来一个新连接，不需要创建一个线程，而是直接注册到clientSelector  SocketChannel clientChannel = ((ServerSocketChannel) key.channel()).accept();  clientChannel.configureBlocking(false);  // OP\_READ表示通道中已经有了可读的数据，可以执行读操作了（通道目前有数据，可以进行读操作了）  clientChannel.register(clientSelector, SelectionKey.OP\_READ);  } finally {  keyIterator.remove();  }  }  }  }  }  } catch (IOException ignored) {  }  }  }.start();  new Thread() {  @Override  public void run() {  String name = Thread.currentThread().getName();  try {  while (true) {  // (2) 批量轮询是否有哪些连接有数据可读，这里的1指的是阻塞的时间为1ms  if (clientSelector.select(1) > 0) {  Set<SelectionKey> set = clientSelector.selectedKeys();  Iterator<SelectionKey> keyIterator = set.iterator();  while (keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();  if (key.isReadable()) {  try {  SocketChannel clientChannel = (SocketChannel) key.channel();  ByteBuffer byteBuffer = ByteBuffer.allocate(1024);  // (3) 读取数据以块为单位批量读取  clientChannel.read(byteBuffer);  byteBuffer.flip();  System.out.println("线程" + name + ":" + Charset.defaultCharset().newDecoder().decode(byteBuffer)  .toString());  } finally {  keyIterator.remove();  key.interestOps(SelectionKey.OP\_READ);  }  }  }  }  }  } catch (IOException ignored) {  }  }  }.start();  }  } |

## 10.3 Netty对NIO的封装

1.添加依赖

|  |
| --- |
| **<dependency>**  **<groupId>io.netty</groupId>**  **<artifactId>netty-all</artifactId>**  **<version>4.1.5.Final</version>**  **</dependency>** |

2、服务端

|  |
| --- |
| public class NettyServer {  public static void main(String[] args) {  ServerBootstrap serverBootstrap = new ServerBootstrap();  NioEventLoopGroup boos = new NioEventLoopGroup();  NioEventLoopGroup worker = new NioEventLoopGroup();  serverBootstrap  .group(boos, worker)  .channel(NioServerSocketChannel.class)  .childHandler(new ChannelInitializer<NioSocketChannel>() {  protected void initChannel(NioSocketChannel ch) {  ch.pipeline().addLast(new StringDecoder());  ch.pipeline().addLast(new SimpleChannelInboundHandler<String>() {  @Override  protected void channelRead0(ChannelHandlerContext ctx, String msg) {  System.out.println(msg);  }  });  }  })  .bind(8000);  }  } |

3.客户端

|  |
| --- |
| public class NettyClient {  public static void main(String[] args) throws InterruptedException {  Bootstrap bootstrap = new Bootstrap();  NioEventLoopGroup group = new NioEventLoopGroup();  bootstrap.group(group)  .channel(NioSocketChannel.class)  .handler(new ChannelInitializer<Channel>() {  @Override  protected void initChannel(Channel ch) {  ch.pipeline().addLast(new StringEncoder());  }  });  Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();  while (true) {  channel.writeAndFlush("测试数据");  Thread.sleep(2000);  }  }  } |

4.Netty的事件通知模型

4.1 轮询方式：就是线程不断轮询访问相关事件发生源有没有发生事件，有发生事件就调用事件处理逻辑

4.2 事件驱动方式：就是线程不去轮询，而是使用消息通知线程来执行处理逻辑

## 10.4 netty和websocket整合的理解

|  |
| --- |
| Websocket作为一种新的通信协议，具备的优势是全双工建立的长端连接，这种连接需要在客户端和服务端实现客户websocket和服务端的websocket，这是应用层的协议，而socket是一个在网络层和传输层的接口，依靠协议来产生两者之间的连接  总的来说websocket协议，为Clientsocket与Serversocket提供了一种全双工长连接的实现方法。  而netty最大的应用场景是高并发状态下的线程管理，当多个连接通过websocket协议创建的服务下来产生数据的交换时，netty可以通过自身的Nio线程管理，来处理这些活动，相当去嵌入到websocket中去来处理websocket的请求与返回数据，达到效率的最大化 |

# 11.框架的整合

## 11.1概述

框架的整合，就是获取到各个框架中自己想要的部分，来进行组装，其实框架也是各个功能模块的集成

## 11.2获取spring容器

|  |
| --- |
| **package** com.tensquare.notice.config;  **import** org.springframework.beans.BeansException; **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.ApplicationContextAware; **import** org.springframework.stereotype.Component;  @Component **public class** ApplicationContextProvider **implements** ApplicationContextAware ***{*** */\*\*  \* 上下文对象实例  \*/* **private static** ApplicationContext *applicationContext*;   @Override  **public void** setApplicationContext***(***ApplicationContext applicationContext***)* throws** BeansException ***{* this**.*applicationContext* = applicationContext;  ***}*** */\*\*  \* 获取applicationContext  \*  \** ***@return*** *\*/* **public static** ApplicationContext getApplicationContext***() {* return** *applicationContext*;  ***}*** */\*\*  \* 通过name获取 Bean.  \*  \** ***@param name*** *\** ***@return*** *\*/* **public** Object getBean***(***String name***) {* return** *getApplicationContext****()***.getBean***(***name***)***;  ***}*** */\*\*  \* 通过class获取Bean.  \*  \** ***@param clazz*** *\** ***@param <T>*** *\** ***@return*** *\*/* **public *<***T***>*** T getBean***(***Class***<***T***>*** clazz***) {* return** *getApplicationContext****()***.getBean***(***clazz***)***;  ***}*** */\*\*  \* 通过name,以及Clazz返回指定的Bean  \*  \** ***@param name*** *\** ***@param clazz*** *\** ***@param <T>*** *\** ***@return*** *\*/* **public *<***T***>*** T getBean***(***String name, Class***<***T***>*** clazz***) {* return** *getApplicationContext****()***.getBean***(***name, clazz***)***;  ***} }*** |

# 12.jackson的使用

## 12.1添加依赖

|  |
| --- |
|  |

## 12.2 pojo类

|  |
| --- |
| public class XwjUser implements Serializable {  private static final long serialVersionUID = 1L;  private int id;  private String message;  private Date sendTime;  // 这里手写字母大写，只是为了测试使用，是不符合java规范的  private String NodeName;  private List<Integer> intList;  public XwjUser() {  super();  }  public XwjUser(int id, String message, Date sendTime) {  super();  this.id = id;  this.message = message;  this.sendTime = sendTime;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getMessage() {  return message;  }  public void setMessage(String message) {  this.message = message;  }  public Date getSendTime() {  return sendTime;  }  public void setSendTime(Date sendTime) {  this.sendTime = sendTime;  }  public String getNodeName() {  return NodeName;  }  public void setNodeName(String nodeName) {  NodeName = nodeName;  }  public List<Integer> getIntList() {  return intList;  }  public void setIntList(List<Integer> intList) {  this.intList = intList;  }  @Override  public String toString() {  return "XwjUser [id=" + id + ", message=" + message + ", sendTime=" + sendTime + ", intList=" + intList + "]";  }  } |

## 13.3 jackson对数据的操作以及还原

|  |
| --- |
| **package** ObjectMapperTest;  **import** com.fasterxml.jackson.core.JsonProcessingException; **import** com.fasterxml.jackson.databind.DeserializationFeature; **import** com.fasterxml.jackson.databind.ObjectMapper; **import** com.fasterxml.jackson.databind.SerializationFeature; **import** org.junit.Test; **import** org.omg.CORBA.PUBLIC\_MEMBER;  **import** java.io.IOException; **import** java.text.SimpleDateFormat; **import** java.util.\*;  */\*\*  \* Description:  \* User: HC  \* Date: 2020-04-11-14:01  \*/* **public class** ObjectMapperTest ***{* public static** ObjectMapper *mapper* = **new** ObjectMapper***()***;   **static *{*** *// 转换为格式化的json  mapper*.enable***(***SerializationFeature.***INDENT\_OUTPUT)***;   *// 如果json中有新增的字段并且是实体类类中不存在的，不报错  mapper*.configure***(***DeserializationFeature.***FAIL\_ON\_UNKNOWN\_PROPERTIES***, **false*)***;  ***}*** */\*\*  \* 对象与json字符串、byte数组  \*  \** ***@throws*** *Exception  \*/* @Test  **public void** testObj***()* throws** Exception ***{*** XwjUser user = **new** XwjUser***(***1, **"Hello World"**, **new** Date***())***;  *// mapper.writeValue(System.out, user); //写到控制台* String jsonStr = *mapper*.writeValueAsString***(***user***)***;  System.***out***.println***(*"对象转化为字符串："** + jsonStr***)***;   **byte*[]*** byteArr = *mapper*.writeValueAsBytes***(***user***)***;  System.***out***.println***(*"对象转为byte数组："** + byteArr***)***;   XwjUser userDe = *mapper*.readValue***(***jsonStr, XwjUser.**class*)***;  System.***out***.println***(*"json字符串转化为对象："** + userDe***)***;   XwjUser useDe2 = *mapper*.readValue***(***byteArr, XwjUser.**class*)***;  System.***out***.println***(*"byte数组转为对象："** + useDe2***)***;   ***}*** */\*\*  \* list集合与json字符串  \* 注意：对象转json字符串时，对象中的NodeName首字母是大写，转出来是小写  \*  \** ***@throws*** *JsonProcessingException  \*/* @Test  **public void** test2***()* throws** JsonProcessingException ***{*** List***<***XwjUser***>*** userList = **new** ArrayList***<>()***;  userList.add***(*new** XwjUser***(***1, **"aaa"**, **new** Date***()))***;  userList.add***(*new** XwjUser***(***2, **"bbb"**, **new** Date***()))***;  userList.add***(*new** XwjUser***(***3, **"ccc"**, **new** Date***()))***;  userList.add***(*new** XwjUser***(***4, **"ddd"**, **new** Date***()))***;   String jsonStr = *mapper*.writeValueAsString***(***userList***)***;  System.***out***.println***(*"集合转为字符串："** + jsonStr***)***;   List***<***XwjUser***>*** userListDes = *mapper*.readValue***(***jsonStr, List.**class*)***;  System.***out***.println***(*"字符串转集合："** + userListDes***)***;  ***}*** *//map与json字符串* @Test  **public void** testMap***()* throws** JsonProcessingException ***{*** HashMap***<***String, Object***>*** testMap = **new** HashMap***<>()***;  testMap.put***(*"name"**, **"jack"*)***;  testMap.put***(*"age"**, 30***)***;  testMap.put***(*"date"**, **new** Date***())***;  testMap.put***(*"user"**, **new** XwjUser***(***1, **"Hello World"**, **new** Date***()))***;   String jsonStr = *mapper*.writeValueAsString***(***testMap***)***;  System.***out***.println***(*"Map转为字符串："** + jsonStr***)***;   Map***<***String, Object***>*** testMapDeco = *mapper*.readValue***(***jsonStr, Map.**class*)***;  System.***out***.println***(*"字符串转mapper："** + testMapDeco***)***;  ***}*** *//修改转换时的日期格式：* @Test  **public void** testOther***()* throws** IOException ***{*** *// 修改时间格式  mapper*.setDateFormat***(*new** SimpleDateFormat***(*"yyyy-MM-dd HH:mm:ss"*))***;  XwjUser user = **new** XwjUser***(***1, **"Hello World"**, **new** Date***())***;  user.setIntList***(***Arrays.*asList****(***1, 2, 3***))***;   String jsonStr = *mapper*.writeValueAsString***(***user***)***;  System.***out***.println***(*"对象转为字符串："** + jsonStr***)***;  ***} }*** |

# 13.Redis分布式缓存

## 13.1 单机模式

## 13.2 主从复制

## 13.3 哨兵模式

## 13.4 cluster内置集群

## 13.5 twemproxy集群

## 13.6使用及其区别

|  |
| --- |
| \* 单机版：数据量，QPS不大的情况使用  \* 主从复制：需要读写分离，高可用的时候使用  \* Sentinel哨兵：需要自动容错容灾的时候使用  \* 内置集群：数据量比较大，QPS有一定要求的时候使用，但集群节点不能过多  \* twemproxy集群：数据量，QPS要求非常高，可以使用 |

# 14. JUC（java.util.current）多线程

## 14.1创建线程的方法

|  |
| --- |
| 无论是继承Thread,还是实现Runnable方法，这些拥有开启线程功能的类都需要实现run方法，线程的核心是对资源的充分利用  存在的问题：主要是争夺资源之时，数据存在不确定性  线程和进程：其实本质上是对内存的划分粗细不一样，进程代表一个运行的程序，他们之间是相互隔离的；线程是对进程的进一步细化，他们之间是可以进行通信的； |

### 1.继承Thread方法

|  |
| --- |
| **public class** Demo1CreateThread ***{* public static void** main***(***String***[]*** args***) {*** System.***out***.println***(*"-----多线程创建开始-----"*)***;  *//1.创建一个线程* CreateThread createThread1= **new** CreateThread***()***;  CreateThread createThread2= **new** CreateThread***()***;  *// 2.开始执行线程 注意 开启线程不是调用run方法，而是start方法* System.***out***.println***(*"-----多线程创建启动-----"*)***;  createThread1.start***()***;  createThread2.start***()***;  System.***out***.println***(*"-----多线程创建结束-----"*)***;  ***}* private static class** CreateThread **extends** Thread ***{*** @Override  **public void** run***(){*** String name = Thread.*currentThread****()***.getName***()***;  **for *(*int** i = 0; i < 5; i++***) {*** System.***out***.println***(***name + **"打印内容是:"** + i***)***;  ***}  }  } }*** |

### 2.实现Runnable方法

|  |
| --- |
| **public static void** main***(***String***[]*** args***) {*** System.***out***.println***(*"-----多线程创建开始-----"*)***;  *//1.创建线程* CreateRunnable createRunnable=**new** CreateRunnable***()***;  Thread thread1 = **new** Thread(createRunnable);  Thread thread2 = **new** Thread(createRunnable);  *// 2.开始执行线程 注意 开启线程不是调用run方法，而是start方法* System.***out***.println***(*"-----多线程创建启动-----"*)***;  thread1.start***()***;  thread2.start***()***;  System.***out***.println***(*"-----多线程创建结束-----"*)***;  ***}* private static class** CreateRunnable **implements** Runnable***{*** @Override  **public void** run***() {*** String name = Thread.*currentThread****()***.getName***()***;  **for *(*int** i = 0; i < 5; i++***) {*** System.***out***.println***(***name + **"的内容:"** + i***)***;  ***}  }  } }*** |

### 3. 匿名内部类方式

|  |
| --- |
| **public class** Demo3Runnable ***{* public static boolean** *exit* = **true**;   **public static void** main***(***String***[]*** args***)* throws** InterruptedException ***{* new** Thread(**new** Runnable() {  @Override  **public void** run() {  String name=Thread.currentThread().getName();  **for** (**int** i = 0; i < 5; i++) {  System.out.println(name + **"执行内容："** + i);  }  }  }).start***()***;   **new** Thread(**new** Runnable() {  @Override  **public void** run() {  String name = Thread.currentThread().getName();  **for** (**int** i = 0; i < 5; i++) {  System.out.println(name + **"执行内容："** + i);  }  }  }).start***()***;   Thread.*sleep****(***1000l***)***;  ***} }*** |

## 14.2用户线程和守护线程

|  |
| --- |
| Java中有两种线程，一种是用户线程，另一种是守护线程。  用户线程是指用户自定义创建的线程，主线程停止，用户线程不会停止  守护线程当进程不存在或主线程停止，守护线程也会被停止。   * 不是守护线程就是用户线程 * 当是守护线程时，主线程停止，守护线程也会停止 * 当是用户线程时，主线程停止，用户线程不会停止   **public class** Demo4Daemon ***{* public static void** main***(***String***[]*** args***)* throws** InterruptedException ***{*** Thread thread = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** i = 0; i < 10; i++) {  **try** {  Thread.sleep(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.out.println(**"子线程..."** + i);  }  }  });   *// 设置线程为守护线程* thread.setDaemon***(*true*)***;  thread.start***()***;   **for *(*int** i = 0; i < 5; i++***) {*** Thread.*sleep****(***10***)***;  System.***out***.println***(*"主线程"** + i***)***;  ***}*** System.***out***.println***(*"主线程执行完毕!"*)***;  ***} }*** |

## 14.3线程安全

## 1.概述

|  |
| --- |
| 1. 程需要考虑时间片的问题  2.线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写  操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，  否则的话就可能影响线程安全。 |

## 2. 线程同步

|  |
| --- |
| synchronized都是对象锁 |

### 2.1.同步代码块

|  |
| --- |
| Object lock = new Object(); //创建锁  synchronized(lock){       //可能会产生线程安全问题的代码  } |

### 2.2同步方法

|  |
| --- |
| //同步方法 this对象  public synchronized void method(){     //可能会产生线程安全问题的代码  }  1.synchronized(lock)，对lock加锁  2.同步方法：this锁  3.静态同步方法：当前类的class对象  会自动判断是否是同一把锁，如果是同一把锁，则同一时间只能一个人获取， |

### 2.3 Lock锁

|  |
| --- |
| Lock lock = new ReentrantLock();  lock.lock();  //需要同步操作的代码  lock.unlock(); |