# 一．Java基础

## 1.0 io与nio

### 1.01 io分类与基础使用

主要分为字节流，字符流，对象流等；字节流是以继承inputStream,outputStream两抽象类的子类如：FileInputStream,BufferedInputStream,FileOutputStream,BufferedOutputStream等；

字符流是以继承Reader,Writer两抽象类的子类如：FileReader,BufferedReader, InputStreamReader

FileWriter,BufferedWriter,OutputStreamWriter等。

使用场景：1.io流是万能流，能传输任意文件； 2.字符流的使用仅仅局限于文本文件，且字符流在处理文件的时候对编码方式的要求严格

Io流的使用注意事项：1.使用流要注意使用完释放资源即关闭连接；2.流的关闭顺序也很重要，理论上是先打开的先关闭，a依赖b，就先关闭a; 3.使用输出流的时候文件不存在，程序会自己创建文件，但不会自己创建文件夹，需要手动去创建文件夹； 4.io不是并发的流，是阻塞的流，某一时刻要么执行读操作要么执行写操作！

### 1.02 nio非阻塞流

概述：面向缓冲的非阻塞流。

核心：通道，缓冲，选择器，非阻塞io

通道类型：FileChannel,DatagramChannel(udp读写网络数据),SocketChannel（tcp）,ServerSocketChannel

缓冲：CharBuffer DoubleBuffer IntBuffer LongBuffer ByteBuffer ShortBuffer FloatBuffer

选择器：适用于单个线程处理多个通道的监听

## 1.1 线程

### 1.1.1创建线程的方式

1. 直接继承Thread（缺点：单继承不易扩展）
2. 实现Runable接口（有点：可以实现多个线程共享资源）常见！！

### 1.1.2线程状态

1.创建：new后的线程

2.就绪：start()的线程，随时可以被cpu调度执行

3.运行：线程获得cpu执行权，只能由就绪状态到运行状态

4.阻塞：某种原因线程放弃cpu执行权，阻塞的情况有三种

a.等待阻塞，调用wait()方法，等待唤醒

b.同步阻塞, 线程在获取synchronized同步锁失败

c.其他阻塞，sleep(),join()等

5.死亡：run完毕或被Kill掉

### 1.1.3 多线程如何避免死锁

1.死锁例子：

**public void** mthod1(){  
 **synchronized** (String.**class**){  
 System.***out***.println(**"一线程获得string的锁，并锁住"**);  
 **synchronized** (Integer.**class**){  
 System.***out***.println(**"线程获得Integer的锁"**);  
 }  
 }  
 System.***out***.println(**"该线程已经执行完程序，没有锁住！"**);  
}  
**public void** method2(){  
 **synchronized** (Integer.**class**){  
 System.***out***.println(**"一线程获得Integer的锁，并锁住"**);  
 **synchronized** (String.**class**){  
 System.***out***.println(**"线程获得Stringd的锁"**);  
 }  
 }  
}

1. 解决死锁：不让线程之间相互占据所需要的锁对象

## 1.2线程通信

1. **synchronized同步，这种方式，本质上就是“共享内存”式的通信。多个线程需要访问同一个共享变量，谁拿到了锁（获得了访问权限），谁就可以执行。**
2. **wait/notify机制**

### 1.2.1 volatile synchronized Lock ReentrantLock

1. volatile和synchronized的区别
2. 锁提供了两种主要特性：互斥（一线程一次只能持有一个特定的锁）和可见性（锁释放之前对共享数据的操作是可见的）
3. 都是为了保证多线程读取数据的一致性
4. Volatrile本质是告诉jvm当前变量的值不确定，需要从主存中读取，synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞
5. Volatile仅使用在变量级别，synchronized可以使用在变量，方法
6. Volatile不会造成线程的阻塞，而synchronized可能会造成线程的阻塞
7. Volatile不具备原子性，不适合在对该变量的操作依赖于自己如n++

### 1.2.2 可重入锁 ReentrantLock

*ReentrantLock逼格满满，完全可以替代Synchronized，可中断响应、锁申请等待限时、公平锁*

1. 乐观锁：每次访问数据线程都是运行状态，提高数据吞吐量，实现方式是CAS
2. 悲观锁 ：每次访问数据线程都是处于阻塞状态，提高了安全性。但有缺点：1.可能会导致优先级高的线程等待优先级低的线程导致优先级倒序；2.加锁，放锁次数过多，引起性能问题
3. 阻塞锁：让线程进入阻塞状态等待，当获得相应的信号（唤醒，时间）时，才可以进入线程的准备就绪状态

## 1.3 集合

### 1.3.1 Linkedhashmap与hashmap的区别

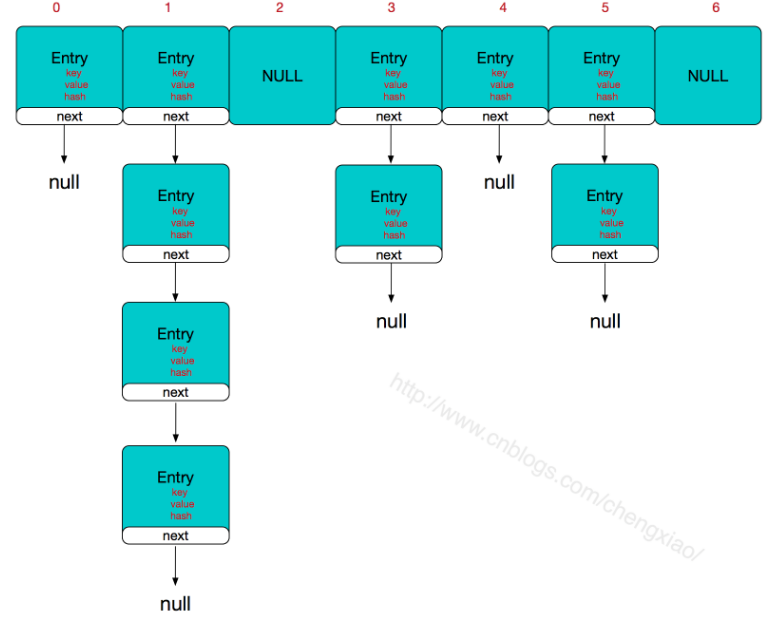
>1.hashMap的结构与原理：

a.hashMap是由数组（主体）+单向链表，数组中存放着entity

b.entity中存放着key-value，hash和指向下一个实体的指针

c.元素是由hash函数计算存储地址放到数组中的

d．哈希冲突：两个同元素通过hash函数得到的实际存储地址一样，hashMAP采用链地址法解决冲突。



>LinkedHashMap是在hashMap的基础上增加了顺序（数组+双向链表）

## 1.4 面向对象的几大特征

>1.抽象：忽略一个主题中与当前目标无关的东西,专注的注意与当前目标有关的方面

>2.封装：把过程和数据包围起来,对数据的访问只能通过特定的界面

>3.继承：为了重用父类代码,同时为实现多态性作准备

>4.多态：是指允许不同类的对象对同一消息做出响应,如子类对父类的重载，同一类中方法的重写

## 1.5 反射（框架设计的灵魂）

>1.简述：在运行状态中，对任意一个类中，都能获得这个类的属性和方法，通俗点讲，反射就是把java类中的各种成分映射成一个个的Java对象。通过反射，该类对我们来说是完全透明的，想要获取任何东西都可以。

>2.反射先得获得类的class（字节码），获得字节码的方法有三种，Class clazz1 = Class.forName("全限定类名");Class clazz2  = Person.class; Class clazz3 = p.getClass();

>3.jdk代理，cglib代理就用到了反射

## 1.6 枚举

>1.什么是枚举？枚举是jdk5以后得新的类型

>2.枚举的格式？public enum Color{RED(“red”,1),BLUE(“blue”,2),YELLOW(“yellow”,3);public String color;public int index;setter,getter…}

>3.枚举的用途？a.可以用在controller里状态码

## 1.7 设计模式

>创建型模式 5种:工厂方法模式，抽象工厂模式，单例模式，建造者模式，原型模式

>结构型模式7种：适配器模式，装饰器模式，代理模式，外观模式，桥接模式，组合模式，享元模式

>行为型模式11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

## 1.8 代理

## 1.9泛型

# 二．javaWeb专题

## 2.1 jdbc dbutil c3p0

## 2.2 过滤器

## 2.3 xml

## 2.4 tomcat

## 2.5 json

## 2.6 jetty

# 三．HTTP TCP UPD 专题

## 3.1 HTTP

### 3.1.1 http请求报文&http请求响应头

>request:请求行(get/url HTTP1.1)，请求头（headrs如User-Agent：xx Accept：xx），消息主体(post提交的参数)

>http响应（http报文头）:状态行（HTTP-version Status-code ok?），消息报头(Date:,Content-Type:,Content-Length:)，响应正文

### http请求方法

>get,post,put,head

### 3.1.3 http请求过程

>1.建立tcp连接🡪🡪2.浏览器向服务器发送请求命令（GET /URL HTTP1.0）🡪🡪3.浏览器发送请求头信息🡪🡪4.服务器应答🡪🡪5.服务器向浏览器发送应答消息报头🡪🡪6.服务器发送响应正文🡪🡪服务器关闭tcp连接

### 3.1.4 GET post put区别

>get 将参数写在url后面，有大小限制，不安全

>post 将参数封装在请求数据中，以Key-value的形式，数据大小没有限制

### 3.1.5 http状态码

>1xx:只是信息—表示已经接收，继续处理

>2xx：成功—表示已经接收，处理

>3xx：重定向—要完成请求必须进一步操作

>4xx：客户端错误—请求资源不存在

>5xx:服务端错误—服务器未能实现合法的请求

### 3.1.6长连接 短连接，http协议是无状态

>短连接：http1.0默认采取短连接，每次访问一个css,jps,html等资源都重新建立连接-传输数据-关闭连接。耗资源

>长连接：http1.1默认采取长连接，Connection:keep-alive,在每次服务器与客户端建立连接后不主动去关闭连接。

>http协议无状态：是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。无状态连接严重阻碍了应用程序的实现，因此想要保持连接状态的技术应运而生，Cookie和session

### 3.1.7转发和重定向

>转发：是服务器的行为，浏览器的地址信息不变

>重定向:是客户端的行为，浏览器的地址发生变化

## 3.2 TCP

>

### 3.2.1 TCP头部

### 3.2.2 tcp三次握手

### 3.2.3 tcp如何保证可靠性传输

### 3.2.4 ip报文 地址类别 私有地址

### 3.2.5网络攻击 dns欺骗 sql注入 ddos攻击

### 3.2.6 socket编程

### 3.2.7 https ssl

## 3.3 UDP

# 四．数据库专题

## 4.1 sql语句

## 4.2 索引

## 4.3 分库分表

### 4.3.1 概念与垂直水平分表

### 4.3.2 如何解决分库分表带来的坏处

## 4.4 数据库锁

### 4.4.1 封锁，封锁协议

### 4.4.2 死锁 活锁

## 4.5 优化

# 五．Java框架专题

## 5.1 spring

### 5.1.1搭建并理解各配置

### 5.1.2 aop ioc di

### 5.1.3解读源码，是如何做到解读xml实现反射等功能

## 5.2 springmvc

### 5.2.1搭建并理解各配置

### 5.2.2解读源码

## 5.3 mybatis

### 5.3.1搭建并理解各配置

### 5.3.2 generator一键生成

### 5.3.3清楚Po-dao-mapper之间映射

### 5.3.4 sql

## 5.4 springboot

## 5.5 springcloud

## 5.6 安全框架shiro

# 六．分布式，缓存专题

## 6.1 dubbo微服务

## 6.2 zookeeper

## 6.3 redis缓存

### 6.3.1 redis的基础知识

A. Redis可以存储键与5种不同数据结构类型之间的映射，这5种数据结构类型分别为String（字符串）、List（列表）、Set（集合）、Hash（散列）和 Zset（有序集合）

B. >1..redis支持16个数据库实例，数据库实例角标0-15，默认登录0号数据库,登录其他数据库语法：select index

>2.事务:开启事务：multi 提交事务：exec 回滚事务：discard 监听事务：watc（乐观锁）

c.redis持久化:

>1.rdb（默认的）： 缺省情况下，Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开redis.conf文件之后，我们搜索save，可以看到下面的配置信息：  
    save 900 1         #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存照。  
    save 300 10      #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存照。  
    save 60 10000   #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内照。

>2.aof（性能低于rdb）

### 6.3.2 redis在项目中的使用(redistemplate)

a.在spring项目中用redistemplate模板（就我而言），RedisTemplate位于spring-data-redis包下，如何使用？🡺

>1.创建redis.properties和redis.xml

>2.redis.xml里注入

<!—连接池-->

<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">  
 <property name="maxIdle" value="${redis.maxIdle}"/>  
 <property name="maxWaitMillis" value="${redis.maxWait}"/>  
 <property name="testOnBorrow" value="${redis.testOnBorrow}"/>  
</bean>

<!-- redis连接工厂 -->  
<bean id="connectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">  
 <property name="poolConfig" ref="poolConfig"/>  
 <property name="port" value="${redis.port}"/>  
 <property name="hostName" value="${redis.host}"/>  
 <property name="password" value=""/>  
 <property name="timeout" value="${redis.timeout}"></property>  
</bean>

<!—redisTemplate模板-->

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">  
 <property name="connectionFactory" ref="connectionFactory"/>  
 <!--序列化key-->  
 <property name="keySerializer">  
 <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>  
 </property>  
 <!--序列化value-->  
 <property name="valueSerializer">  
 <bean class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer"/>  
 </property>  
</bean>

<!—redisutil看情况吧，一般是要配合util使用-->

<bean id="redisUtil" class="com.github.Duankan.utils.RedisUtil">  
 <property name="redisTemplate" ref="redisTemplate"/>  
</bean>

>3.redisTemplate的使用总结：增删查改（序列化操作，放取都需序列化）, redisTemplate定义了五种数据结构操作（具体操作百度）：

redisTemplate.opsForValue();//操作字符串

redisTemplate.opsForHash();//操作hash

redisTemplate.opsForList();//操作list

redisTemplate.opsForSet();//操作set

redisTemplate.opsForZSet();//操作有序set

ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();//序列化操作，放取都需序列化  
operations.set(key, value);

Operations.get(key);

### 6.3.3 redis的几种主流序列化与反序列化

1.jdk序列化与反序列化

//jdk序列化  
public static byte[] serializer\_jdk(Object object){  
 ObjectOutputStream oos=null;  
 ByteArrayOutputStream baos=null;  
 try{  
 baos=new ByteArrayOutputStream();  
 oos=new ObjectOutputStream(baos);//包装模式？  
 oos.writeObject(object);  
 byte[] bytes=baos.toByteArray();  
 return bytes;  
 }  
 catch (Exception e){  
 e.printStackTrace();  
 }  
 return null;  
}  
//jdk反序列化  
public static Object unSerializer\_jdk(byte[] bytes){  
 ObjectInputStream ois=null;  
 ByteArrayInputStream bais=null;  
 try{  
 bais=new ByteArrayInputStream(bytes);  
 ois=new ObjectInputStream(bais);  
 return ois.readObject();  
 }  
 catch (Exception e){  
 e.printStackTrace();  
 }  
 return null;  
}

2.protostuff序列化与反序列化

3.kryo序列化与反序列化

## 6.4 elasticsearch搜素引擎

## 6.5 hadoop大数据

## 6.6 nginx负载均衡

## 6.7 docker

# 七．事务，队列，工作流等专题

## 7.1 Jdbc事务

## 7.2 ssm框架事务

## 7.3分布式消息中间件

## 7.3.1 rabitmq

## 7.3.2 activemq

## 7.4工作流：activity

# 八．linux专题

# 九．jvm db调优专题

# 十．前端专题

## 10.1 vue ivew

## 10.2 js高级

## 10.3 jquery

## 10.4 bootstrap layer

## 10.5 css html布局

# 十一.开发工具专题

## 11.1 idea

## 11.2 git

## 11.3 maven

## 11.4 mysql

## 11.5 svn

## 11.6 tomcat

## 11.7 linux虚拟机

## 11.8 vscode