**Java知识整理**

1. **面向对象的思想**
   1. **基本概念**
      1. **封装**

将客观事物封装成抽象的类，这些类可以把自己的数据与方法让可信的类或对象操作，对不可信的类进行信息隐藏。

* + 1. **继承**

使用现有类的所有的功能，在无需重新编写原来的类的情况下对这些功能进行拓展。Java中只支持类的单继承性。一个类只能有一个直接的父类。

* + 1. **多态**

一个类的实例的相同方法在不同情况下有不同的表现形式。最常见的多态就是将子类传入父类参数中，运行时调用父类方法时通过传入的子类决定具体的内部结构或行为。（子类B继承父类A，声明使用父类A而创建使用子类B）

实现多态的三个条件

* 继承
* 重写（子类对象重写父类的方法）
* 父类引用指向子类对象

多态在父子类中的方法存在的三种情况：

1. 父类中有而子类中没有的方法，使用多态时会直接调用父类的方法。
2. 父类中有而子类中对父类的方法进行了重写，这时候调用的时子类的方法。
3. 父类中没有而子类中有的方法，这时调用会出错。
   1. **设计原则**
      1. **单一职责原则**

类的功能要单一。

* + 1. **里氏替换原则**

子类可以替换父类，出现在父类出现的任何地方（多态）。

* + 1. **依赖倒转原则**

高层次模块不应该依赖低层次的模块，他们应该依赖于抽象，抽象不应该依赖于具体实现，具体实现应该依赖于抽象。针对接口编程

* + 1. **迪米特法则（最少知道）**

一个对象应该对其他对象尽可能少的了解，尽可能少的与其他对象之间发生相互作用。

* + 1. **聚合复用原则**

对象委托优于继承，在一个新的对象中尽量通过持有其他对象来整合功能，避免使用继承。

* + 1. **开放封闭原则**

对拓展开放，对修改封闭。

* + 1. **接口隔离原则**

采用多个于特定客户类有关的接口比采用一个通用的接口好。接口功能要单一。降低类之间的耦合与依赖。

* 1. **UML**图
     1. 表示一个类

用三层矩形结构来表示一个类



第一层为类名（斜体表示抽象类）

第二层为属性名（+表示public，-表示private，#表示protected）

第三层为成员方法（访问权限与属性相同）

* + 1. 类与类之间的**继承**的关系

空心三角形加实线表示



* + 1. 类与类之间的依赖关系

带有箭头的虚线表示



* + 1. 类与类之间的关联关系

带有箭头的实线表示



* + 1. 表示一个接口

在类的三层结构类名处加上<<interface>>

接口实现使用虚线加空心三角形



* + 1. 表示聚合关系

空心菱形加箭头实线

组合关系

实心菱形加实线箭头



* 1. UML序列图

通过描述对象之间发送消息的时间顺序显示多个对象之间的动态协作

1. **Java重点基础知识**
   1. **java常用类、接口**

* java.lang.Object

所有java类的基类。包含的方法：

Object clone()：克隆一个与该对象相同的类的对象。

要实现一个对象的深复制，必须满足以下三个条件：

1. 被复制的类必须实现java.lang.Cloneable接口，该接口为一个标记接口，没有任何方法（不是实现该接口会报错java.lang.CloneNotSupportedException）
2. 重写Object的clone()方法，调用super.clone()拿到一个新的对象。
3. 被复制对象所持有的其他对象也必须实现Cloneable接口、重写其clone()方法,在被复制对象中对这些对象进行同样的拷贝（调用super.clone()方法）。

例子：

public class CloneTest {  
 public static void main(String[] args) {  
 CloneA cloneA = new CloneA("clonea");  
 CloneB cloneB = new CloneB(3);  
 try {  
 CloneA cloneA1 = (CloneA) cloneA.clone();  
 System.*out*.println(cloneA);  
 System.*out*.println(cloneA1);  
 System.*out*.println(cloneA.getA() == cloneA1.getA());  
 cloneA.setA("test111");  
 System.*out*.println(cloneA.getA());  
 System.*out*.println(cloneA1.getA());  
 System.*out*.println(cloneA.getA() == cloneA1.getA());  
 CloneB cloneB1 = new CloneB(444);  
 CloneA cloneA2 = new CloneA("clone...",cloneB);  
 CloneA cloneA3 = (CloneA) cloneA2.clone();  
 System.*out*.println(cloneA2);  
 System.*out*.println(cloneA3);  
 System.*out*.println(cloneA2.getB());  
 System.*out*.println(cloneA3.getB());  
 } catch (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
class CloneA implements Cloneable{  
 String a;  
 CloneB b;  
 public CloneA(String a) {  
 this.a = a;  
 }  
 public CloneA(String a, CloneB b) {  
 this.a = a;  
 this.b = b;  
 }  
 @Override  
 protected CloneA clone() throws CloneNotSupportedException {  
 CloneA cloneA = (CloneA) super.clone();  
 if(null != b){  
 cloneA.setB((CloneB) (getB().clone()));  
 }  
 return cloneA;  
 }  
 public CloneB getB() {  
 return b;  
 }  
 public void setB(CloneB b) {  
 this.b = b;  
 }  
 public String getA() {  
 return a;  
 }  
 public void setA(String a) {  
 this.a = a;  
 }  
}  
class CloneB implements Cloneable{  
 int a;  
 public CloneB(int a) {  
 this.a = a;  
 }  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
 public int getA() {  
 return a;  
 }  
 public void setA(int a) {  
 this.a = a;  
 }  
}

输出结果;

com.steven.ObjectDemo.CloneA@2ff5659e

com.steven.ObjectDemo.CloneA@77afea7d

true

test111

clonea

false

com.steven.ObjectDemo.CloneA@161cd475

com.steven.ObjectDemo.CloneA@532760d8

com.steven.ObjectDemo.CloneB@57fa26b7

com.steven.ObjectDemo.CloneB@5f8ed237

final getClass()

动态获取对象的运行时实例的类类型。

例子：

package com.steven.ObjectDemo;  
public class ClassTest {  
 public static void main(String[] args) {  
 Fruit fruit = new apple();  
 System.*out*.println(Fruit.class.getName());  
 System.*out*.println(fruit.getClass().getName());  
 }  
}  
class Fruit{  
}  
class apple extends Fruit{  
}

输出结果：

com.steven.ObjectDemo.Fruit

com.steven.ObjectDemo.apple

equals()

在Object中比较的是对象的内存地址

public boolean equals(Object obj) {  
 return (this == obj);  
}

toString()

描述当前对象的相关信息：

public String toString() {  
 return getClass().getName() + "@" + Integer.*toHexString*(hashCode());  
}

可以看到toString中调用了hashCode()方法。

native int hashCode()

返回当前对象的hashCode值，这是一个本地方法。

final native void notify()

唤醒一个等待中的线程。

final native void notifyAll()

唤醒所有等待在该对象上的线程。

public final native void wait()

导致线程进入等待状态，直到被其他线程通过notify或notifyAll方法唤醒。

* java.lang.Thread

synchronized void start()

导致线程开始执行

void run()

编写线程运行业务逻辑的方法

static void sleep(long millis) throws InterruptedException

导致当前正在执行的线程进入睡眠状态，在睡眠状态下的线程不会释放锁。

final void join()

合并线程，当前线程等待被合并线程执行完成后继续执行。

for (int i = 0; i < 40; i++) {  
 if(i == 20){  
 try {  
 t1.join();// t1执行完成后主线程继续执行  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+"-->"+i);  
}

static native yield()

暂停当前线程让出cpu时间，当前线程进入就绪状态。

new Thread(()->{  
 for (int i = 0; i < 30; i++) {  
 System.*out*.println(Thread.*currentThread*().getName()+"-->"+i);  
 }  
},"ThreadA").start();  
for (int i = 0; i < 30; i++) {  
 if(i%5 == 0){  
 Thread.*yield*();  
 }  
 System.*out*.println("main thread"+"-->"+i);  
}

* java.lang.String
* java.lang.StringBuilder/StringBuffer
* java.lang.System
* java.util.Arrays
* java.util.Scanner
  1. **java IO**
* IO的概念：输入输出，关注的是从原始数据源的读取（输入）和输出原始数据到目标媒介，是站在程序的立场看输入输出。
* 数据源与目标媒介的种类：

文件

管道

网络连接

内存

System.in 、System.out、System.error（java的标准输入输出与错误输出）

* 流（Stream）:

一个连续的数据流，流与数据源或数据流向的媒介相关联，可以是处理字节的流，也可以是处理字符的流。

* 文件

java中关于文件的类：

java.io.File 基本文件及目录操作中

java.io. RandomAccessFile 随机读写文件类

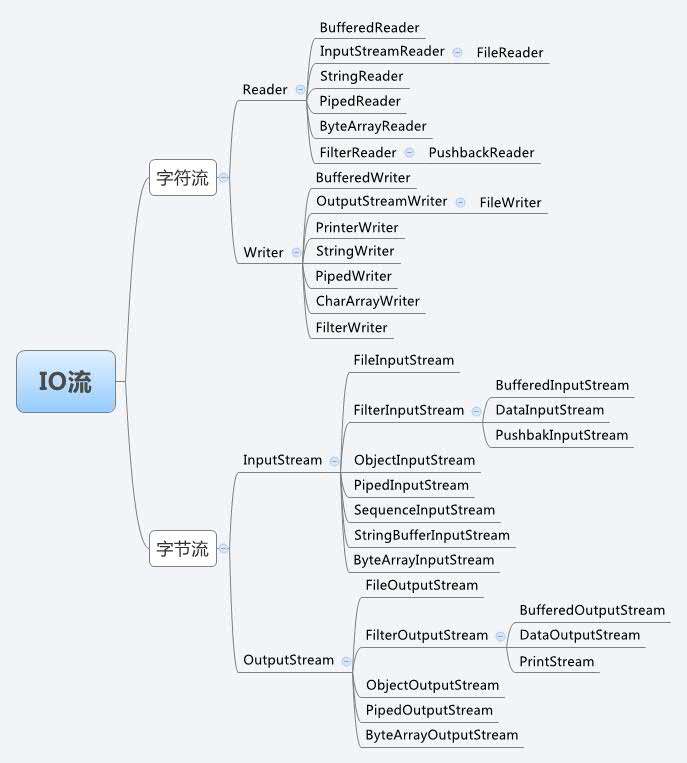
RandomAccessFile随机读写用法：写入文件较大时，使用RandomAccessFile能避免内存溢出。

public void test2() throws IOException {  
 int length = 0x8000000; // 128 Mb  
 // 为了以可读可写的方式打开文件，这里使用RandomAccessFile来创建文件。  
 FileChannel fc = new RandomAccessFile("readtest.txt", "rw").getChannel();  
 //注意，文件通道的可读可写要建立在文件流本身可读写的基础之上  
 MappedByteBuffer out = fc.map(FileChannel.MapMode.*READ\_WRITE*, 0, length);  
 //写128M的内容  
 for (int i = 0; i < length; i++) {  
 out.put((byte) 'x');  
 }  
 System.*out*.println("Finished writing");  
 //读取文件中间6个字节内容  
 for (int i = length / 2; i < length / 2 + 6; i++) {  
 System.*out*.print((char) out.get(i));  
 }  
 fc.close();  
}

java的IO可以分为阻塞（Blockin）IO与非阻塞（non-Blocking）IO.

* + 1. **BIO体系**

体系结构图



阻塞IO的两种分类：

1. 字节流与字符流

区分方式：以Stream结尾的类为字节流，以Reader/Writer结尾的为字符流。

字节流：一次可读入或读出一个字节（八位）的流，能读取所有文件类型

字符流：一次可读入或读出两个字节（16位）的流，字符流不可用于复制图片与视频，会造成文件打不开。

字符流与字节流之间的转换流：InputStreamReader/OutputStreamWriter。

转换流的作用：

将字节流转换为字符流，只能处理纯文本内容

转换处理过程中可以指定字符集。

例子：

//网络流,指定字符集,使用BufferedReader  
@Test  
public void test2() {  
 try (BufferedReader reader =  
 new BufferedReader(  
 new InputStreamReader(  
 new URL("https://www.baidu.com/").openStream(), "UTF-8"));  
 BufferedWriter writer =  
 new BufferedWriter(  
 new OutputStreamWriter(  
 new FileOutputStream("baidu.html"), "UTF-8"))) {  
 String msg = null;  
 while ((msg = reader.readLine()) != null) {  
 System.*out*.print(msg);  
 writer.write(msg);  
 writer.newLine();  
 }  
 writer.flush();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

1. 节点流与处理流

节点流：直接从数据源或目标媒介读写数据的流。

常用节点流：

FileInputStream/FileOutputStream（文件字节输入输出流）ByteArrayInputStream/ByteArrayOutputStream（字节数组输入输出流，面向内存，不要使用多态操作）

CharArrayReader/CharArrayWriter（字符数组读写流，面向内存）

FileReader/FileWriter（文件字符输入输出流）

处理流：不直接处理数据源或目标媒介，对其他流进行封装的流。

常用处理流：

BufferedInputStream/BufferedOutputStream（字节缓冲流）BufferedReader/BufferedWriter、（字符缓冲流，不要使用多态操作）

InputStreamReader/OutputStreamWriter(字节字符转换流)

DataInputStream/DataOutputStream（基本数据类型读写流）

ObjectInputStream/OutputStream(对象输入输出流，序列化和反序列化流，被序列化或反序列化的类必须实现了Serializable接口)

PrintStream(打印流，System.out的类型)、PrintWriter(打印输出)

SequenceInputStream（输入合并流，将多个输入流合并程一个流）

处理流释释放时只需要释放外层处理流，不需要释放里层节点流，JDK会自动释放里层节点流。

处理流使用了装饰器模式。

所有的IO操作都必须使用节点流来操作，处理流提高了操作效率。

组合使用例子：

@Test  
public void test1() {  
 ByteArrayOutputStream bos = new ByteArrayOutputStream();  
 DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(bos));  
 byte[] datas = null;  
 try {  
 dos.writeUTF("床前明月光，疑是地上霜。举头望明月，低头思故乡。");  
 dos.writeInt(27);  
 dos.writeBoolean(true);  
 dos.writeChar(65);  
 dos.writeBytes("hello");  
 dos.writeChars("world");//读出的时候只能一个char一个char的读  
 dos.writeDouble(200.0);  
 dos.flush();  
 datas = bos.toByteArray();  
 String writerinfo = new String(datas);  
 System.*out*.println(writerinfo);  
 System.*out*.println(datas.length);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 DataInputStream dis = new DataInputStream(new BufferedInputStream(new ByteArrayInputStream(datas)));  
 try {  
 //读写数据类型顺序一定要保持一致  
 String msg = dis.readUTF();  
 System.*out*.println(msg);  
 int resultInt = dis.readInt();  
 System.*out*.println(resultInt);  
 boolean resultBoolean = dis.readBoolean();  
 System.*out*.println(resultBoolean);  
 char resultChar = dis.readChar();  
 System.*out*.println(resultChar);  
 byte[] bytes = new byte[5];  
 dis.read(bytes);  
 System.*out*.println(new String(bytes));  
 char c1 = dis.readChar();  
 System.*out*.println(c1);  
 char c2 = dis.readChar();  
 System.*out*.println(c2);  
 char c3 = dis.readChar();  
 System.*out*.println(c3);  
 char c4 = dis.readChar();  
 System.*out*.println(c4);  
 char c5 = dis.readChar();  
 System.*out*.println(c5);  
 System.*out*.println(dis.readDouble());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

* + 1. **NIO**

Non-blocking I/O，同步非阻塞IO，IO多路复用的基础，在高并发，大量连接的场景中使用很多。

BIO中阻塞的地方在于read或write时的阻塞，关注的是“我要读了”，并在此阻塞。

NIO中只要有数据，就可以读，关注的是“我可以读了”。

NIO的主要优势是在网络IO中的使用。

NIO三个重要组件

* buffer 缓冲区，数据缓存的工具

buffer中维护的变量

1. capacity，缓存容纳数据元素的最大数量
2. limit缓冲区里的数据总量
3. position，下一个要被读或写的数据
4. mark，上次读写位置。

方法：

get（）获取buffer中的数据

put（）将数据放到buffer中

flip（）需要读数据时调用该方法切换成读模式

clear（）清除缓存数据，切换成写模式

public static void main(String[] args) {  
 // 创建一个缓冲区  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(1024);  
 // 看一下初始时4个核心变量的值  
 System.*out*.println("初始时-->limit--->"+byteBuffer.limit());  
 System.*out*.println("初始时-->position--->"+byteBuffer.position());  
 System.*out*.println("初始时-->capacity--->"+byteBuffer.capacity());  
 System.*out*.println("初始时-->mark--->" + byteBuffer.mark());  
 System.*out*.println("--------------------------------------");  
  
 // 添加一些数据到缓冲区中  
 String s = "steven lee";  
 byteBuffer.put(s.getBytes());  
  
 // 看一下初始时4个核心变量的值  
 System.*out*.println("put完之后-->limit--->"+byteBuffer.limit());  
 System.*out*.println("put完之后-->position--->"+byteBuffer.position());  
 System.*out*.println("put完之后-->capacity--->"+byteBuffer.capacity());  
 System.*out*.println("put完之后-->mark--->" + byteBuffer.mark());  
  
 byteBuffer.flip();  
 // 看一下初始时4个核心变量的值  
 System.*out*.println("flip完之后-->limit--->"+byteBuffer.limit());  
 System.*out*.println("flip完之后-->position--->"+byteBuffer.position());  
 System.*out*.println("flip完之后-->capacity--->"+byteBuffer.capacity());  
 System.*out*.println("flip完之后-->mark--->" + byteBuffer.mark());  
  
 // 创建一个limit()大小的字节数组(因为就只有limit这么多个数据可读)  
 byte[] bytes = new byte[byteBuffer.limit()];  
 // 将读取的数据装进我们的字节数组中  
 byteBuffer.get(bytes);  
 // 输出数据  
 System.*out*.println(new String(bytes, 0, bytes.length));  
 //读完之后的变量值  
 // 看一下初始时4个核心变量的值  
 System.*out*.println("get完之后-->limit--->"+byteBuffer.limit());  
 System.*out*.println("get完之后-->position--->"+byteBuffer.position());  
 System.*out*.println("get完之后-->capacity--->"+byteBuffer.capacity());  
 System.*out*.println("get完之后-->mark--->" + byteBuffer.mark());  
 byteBuffer.flip();  
 // 看一下初始时4个核心变量的值  
 System.*out*.println("flip完之后-->limit--->"+byteBuffer.limit());  
 System.*out*.println("flip完之后-->position--->"+byteBuffer.position());  
 System.*out*.println("flip完之后-->capacity--->"+byteBuffer.capacity());  
 System.*out*.println("flip完之后-->mark--->" + byteBuffer.mark());  
}

* channel 管道，数据传输的工具，双向传输，不对数据做处理
* selector 选择器，一般在网络多路复用（socket通信中用的多，参考netty）

几个概念

分散读取（scatter）：将一个通道中的数据分散读取到多个缓冲区中

聚集写入（gather）：将多个缓冲区中的数据集中写入到一个通道中

通过Channel管道运输着存储数据的Buffer缓冲区的来实现数据的处理。

* + 1. **开源IO工具**
    2. **AIO**
  1. **Java多线程**
     1. **一些概念**

**进程：**正在运行程序的实例

**线程：**是轻量级进程，可以并行运行并与它们的父进程（即创建它们的进程）共享地址空间（即，一系列内存位置）和其他资源。

**上下文切换：**CPU从一个线程或进程切换到另一个线程或进程。上下文切换是一个十分消耗系统资源的过程。

**并发：**同一时间只能做一个任务，多个任务被快速轮换执行，使得在宏观上有多个任务在同时执行。

**并行：**同一台处理器上同时处理多个任务

**同步：**当一个线程操作内存时，其他线程不允许对该内存地址进行操作，知道该线程完成操作，其他线程才能继续操作，在此期间，其他线程处于等待状态。

**异步：**任务执行无线顺序，一旦开始，方法调用立即返回，调用者继续后面的行为。

**临界资源：**公共共享资源，能被多个线程共享的资源。临界资源在同一时间只能有一个访问线程，其他想使用该资源的线程处于等待状态，直到当前线程释放该资源。

**临界区：**访问临界资源的同步代码。

**阻塞：**当一个线程占用临界资源，其他线程也想使用该资源就必须等待，等待会导致线程挂起进入阻塞状态。

**非阻塞：**所有线程不受其他线程访问临界资源的影响，尝试执行下一步操作。

**死锁：**两个或两个以上的线程执行过程中由于竞争资源或由于彼此通信而造成的一种阻塞现象。

* + 1. **线程状态**

新建（new）：通过new关键字创建线程

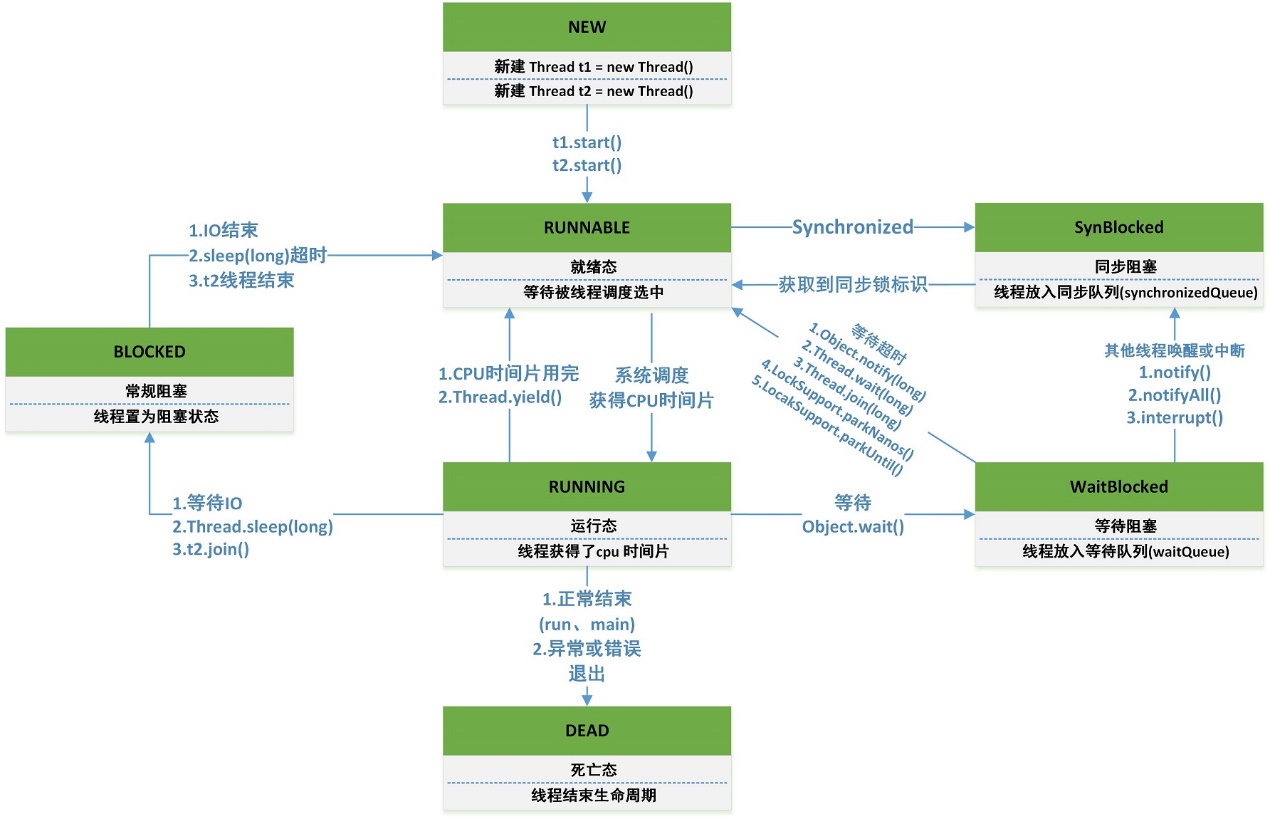
就绪（runnable）：调用Thread的start（）方法且还未获取到cpu时间片，或阻塞时间解除，或调用yeild（）方法

运行（running）：获取到cpu时间片执行Thread的run方法

阻塞（block）：线程因为某种原因放弃cpu时间片暂时停止运行，有三种情况会导致线程阻塞：1.常规阻塞，调用发出IO请求，执行Thead.sleep方法，Thread.join方法；2.同步阻塞：线程获取对象同步锁时，该同步锁已被其他线程占用，线程进入阻塞状态；3.等待阻塞：调用wait方法

死亡（dead）：线程main方法、run方法执行完毕或出现异常，线程死亡。

线程状态转换图（来自网络）



* + 1. **线程基本用法**

**Thread**

**继承Thread类重写run方法，调用start启动**

**Runnable**

**实现Runnable接口重写run方法，将实现类作为参数传递给Thread的构造器，调用start方法**

**Callable**

**实现Callable接口重写call方法，使用Future接收线程执行结果。**

**public class SimpleCallableTest {**

**@Test**

**public void test() throws ExecutionException, InterruptedException {**

**ExecutorService executorService = Executors.newFixedThreadPool(1);**

**SimpleCallable simpleCallable = new SimpleCallable();**

**Future<Integer> result = executorService.submit(simpleCallable);**

**int num = result.get();**

**System.out.println(num);**

**executorService.shutdown();**

**}**

**}**

**class SimpleCallable implements Callable<Integer>{**

**@Override**

**public Integer call() throws Exception {**

**return 100;**

**}**

**}**

* + 1. **死锁例子**

**public class SimleDeadLockTest {**

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

**Object goods = new Object();**

**Object money = new Object();**

**Transection transection = new Transection(goods,money);**

**Transection2 transection2 = new Transection2(goods,money);**

**Thread thread = new Thread(transection);**

**Thread thread2 = new Thread(transection2);**

**thread.start();**

**thread2.start();**

**}**

**}**

**class Transection implements Runnable{**

**Object goods;**

**Object money;**

**public Transection(Object goods, Object money) {**

**this.goods = goods;**

**this.money = money;**

**}**

**public Transection() {**

**}**

**@Override**

**public void run() {**

**while(true){**

**test();**

**}**

**}**

**public void test(){**

**synchronized (goods){**

**System.out.println("goods is locked!");**

**try{**

**Thread.sleep(500);**

**}catch (InterruptedException e){**

**e.printStackTrace();**

**}**

**synchronized (money){**

**}**

**}**

**System.out.println("一手给钱");**

**}**

**}**

**class Transection2 implements Runnable{**

**Object goods;**

**Object money;**

**public Transection2(Object goods, Object money) {**

**this.goods = goods;**

**this.money = money;**

**}**

**public Transection2() {**

**}**

**@Override**

**public void run() {**

**while(true){**

**test();**

**}**

**}**

**public void test(){**

**synchronized (money){**

**System.out.println("money is locked!");**

**try{**

**Thread.sleep(500);**

**}catch (InterruptedException e){**

**e.printStackTrace();**

**}**

**synchronized (goods){**

**}**

**}**

**System.out.println("一手给货");**

**}**

**}**

分析线程状态的工具：jstack

* + 1. **ThreadLocal（解决多线程本地（局部）变量的问题）**

ThreadLocal通过为每个线程提供一个独立的变量副本解决变量并发访问问题。ThreadLocal中的变量副本改变不会影响其他线程的变量副本。

ThreadLocal中维护了一个ThreadLocalMap，其中键使用线程对象，值为变量副本。

ThreadLocal不是为了解决线程共享变量的问题，而是为每个线程创建一个单独的副本变量。

重要方法

set():设置设置当前线程在当前ThreadLocal中的本地变量值。

get()：返回当前线程在ThreadLocal中存储本地变量的值，如果当前值不存在，则返回initialValue方法设置的默认初始值。

remove():移除当前线程在当前ThreadLocal中存储的本地变量值，移除后如果没有执行set操作，get到的都是initialValue方法设置的默认初始值。

initialValue(): 设置变量的初始值，通过get方法在ThreadLocalMap中拿到的变量值为null时，会调用该方法。

* + 1. **synchronized与volatile**

**volatile 保证内存可见性和代码的有序性（禁止指令重排），不能保证操作的原子性，不具备互斥性。**

java.util.concurrent.atomic包下的原子类解决了volatile的不足，原子类具备一下特征：

使用volatile保证变量的内存可见性

使用CAS算法保证原子性

synchronized保证了变量的可见性、原子性和有序性

* + 1. **CAS算法（JUC中大量使用了CAS算法）**

CompareAndSwap比较并交换

三个操作数：

内存地址V

旧的预期值A

即将更新的目标值B

* + 1. **synchronized与lock的区别**

Lock是一个接口，synchronized是一个关键字

synchronized锁定发生异常时会自动释放锁，不会造成死锁现象。Lock发生异常如果没有调用unLock（）方法不会自动释放锁（释放锁的代码要放在finally块中）。

Lock可以让等待锁的线程响应中断去做其他操作，synchronized能响应中断。

通过Lock能知道是否锁定成功，synchronized无法做到

Lock能提高多个线程的读写效率。

synchronized无法中断（interrupt），Lock允许中断

synchronized锁定范围：



synchronized锁定规则：

1. 普通方法调用与同步方法调用互不影响。当一个线程进入一个类的同步方法时，其他线程可以同时访问该类的非同步方法。
2. 所有同步方法同时只能被一个线程访问。当一个线程进入一个类的同步方法时，其他线程无法同时访问该类的其他同步方法。
3. 当同步代码块都是同一个锁时，方法可以被所有线程访问，锁的同步代码块同一时间只能被一个线程访问，且同步代码块的执行顺序不定。
4. 可重入性。当一个线程再次请求自己持有对象锁的临界资源时，请求将会成功，这就是可重入锁。

注意：不要将String对象作为锁定对象，String常量池具有缓存功能，相同字面量时同一个锁。

* + 1. **CAS算法**
    2. **锁**

锁在java中具体表现为Lock和synchronized

Lock标准用法

Lock lock = new ReentrantLock();

//获取锁应在try之前，因为若获取锁时发生异常，异常抛出同时会导致锁无故释放

lock.lock();

try{

doSometing();

}finally{

//注意：必须在finally块中释放锁，目的是保证在获取到锁之后，最终能被释放

lock.unlock();

}

Lock提供的接口：

void lock()：获取锁

void lockInterruptibly() throws InterruptedException：当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

    boolean tryLock()：尝试获取锁，获取成功返回true

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException：尝试在指定时间内获取锁，获取成功返回true

    void unlock()：释放锁

    Condition newCondition()：返回一个等待通知的条件

JUC中的Lock

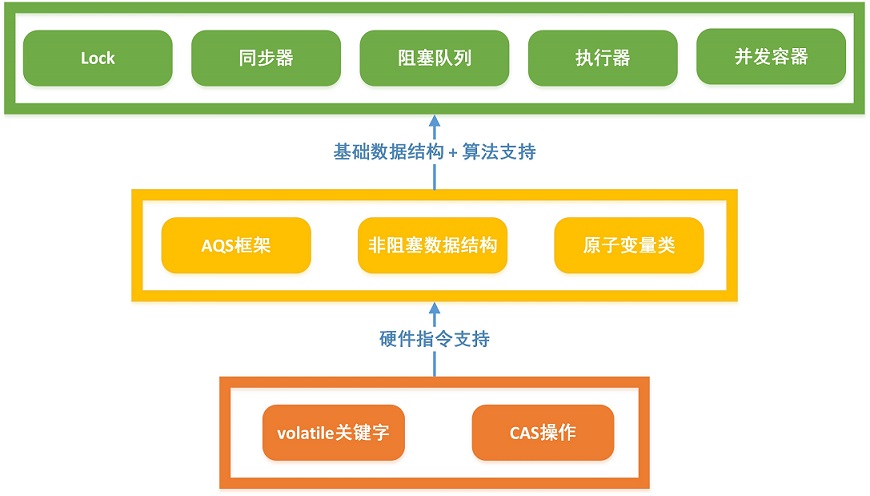
ReentrantLock：可重入锁

ReadWriteLock：读写锁接口

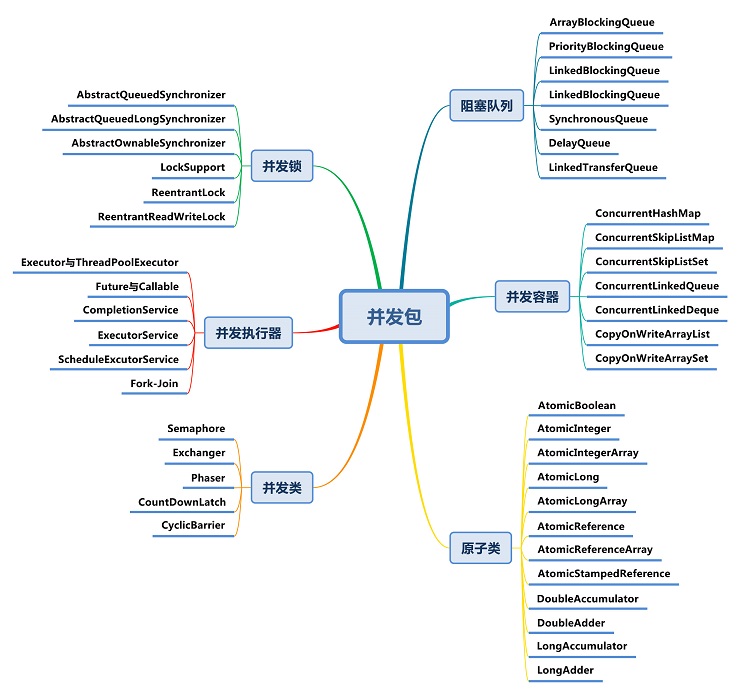
ReentrantReadWriteLock：读写锁实现，readLock()和writeLock()用来获取读锁和写锁。

* + 1. **JUC**

**concurrent包结构**



**concurrent类图**



* + 1. **线程池**
  1. **Java网络编程**
     1. **一些概念**
     2. **Socket基本用法**
     3. **爬虫原理**
     4. **Netty**
  2. **Java集合**
     1. **泛型**
     2. **集合体系结构**
     3. **基本用法**
     4. **java.util.concurrent包下的容器**
  3. **Java异常**
     1. **异常体系结构**

异常：程序运行过程中出现的错误。

Java中所有的异常都继承自抽象类java.lang.Throwable。

子类Error是虚拟机内部错误，如内存溢出错误（OutOfMemoryError）、线程终止错误（ThreadDeath）。这种错误程序无法处理，一般由虚拟机终止进程。

子类Exception是程序可以处理的异常。这种异常分为运行时异常（RuntimeException）和非运行时异常（又称检查异常CheckedException）。运行时异常可以不进行捕获，也可以进行捕获，非运行异常必须进行捕获处理。

常见异常：

ArithmeticException 算数异常

ArrayStoreException         试图将错误类型的对象存储到一个对象数组时抛出的异常

NullPointerException      空指针异常

NumberFormatException 数值强转异常

IndexOutOfBoundsException  数组下标越界异常

IllegalArgumentException   抛出的异常表明向方法传递了一个不合法或不正确的参数



* + 1. **异常处理**

java中的异常处理有两种方式：

1. 使用try{}catch(Exception e){}finally{}语法对异常进行捕获

程序首先进入try语句执行，如果没有异常则执行return之前会先执行finally中的语句（如果finally中存在return则不会再执行try中的return）。

try catch语法糖：

//使用try-with-resource不再需要手动的去关闭流  
try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.*in*));  
 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(System.*out*))) {  
 String msg = "";  
 while (!msg.equals("exit")) {  
 msg = reader.readLine();  
 writer.write(msg);  
 writer.newLine();  
 writer.flush();  
 }  
} catch (IOException e) {  
 System.*out*.println("exception");  
}

1. 在方法签名上使用throws将异常抛出，由调用者处理。

throw与throws的区别

throw 用于在方法体中抛出异常

throws 用在方法签名上，用来标识该方法可能抛出异常。

* + 1. **异常处理原则**
* 避免使用异常处理代替错误处理
* 处理异常不可代替简单测试
* 不要进行小粒度的异常处理，影响程序可读性
* 异常往往在高层处理，由调用者处理，层层外抛
  1. **Java反射**
     1. **反射的作用**
     2. **反射的基本用法**
     3. **自定义注解**
     4. **JDK与CGLIB动态代理**
     5. **Java动态编译**
  2. **JVM**
     1. **JVM内存模型**
     2. **Java类加载机制**
     3. **垃圾回收**
     4. **JVM监控工具**
     5. **JVM优化**
  3. **其他**
     1. **关键字**

native:调用本地方法

class:类字面量

* + 1. **访问控制范围**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 关键字  范围 | public | protected | default(默认) | private |
| 不同包 |  |  |  |  |
| 同包 |  |  |  |  |
| 子类 |  |  |  |  |
| 自身 |  |  |  |  |

* + 1. **==与equals**

java中==比较的是两个对象的内存地址，java.lang.Object中的equals方法实现如下：

public boolean equals(Object obj) {  
 return (this == obj);  
}

即Object中的equals和==是等价的。

对于基本数据类型int、short、long、boolean、byte、float、double、char，其包装类Integer、Short、Long、Boolean、Byte、Float、Double、Character对equals方法进行了重写。Integer中equals的实现;

public boolean equals(Object obj) {  
 if (obj instanceof Integer) {  
 return value == ((Integer)obj).intValue();  
 }  
 return false;  
}

Float中的equals实现：

public boolean equals(Object obj) {  
 return (obj instanceof Float)  
 && (*floatToIntBits*(((Float)obj).value) == *floatToIntBits*(value));  
}

结论：基本数据类型的equals比较的是变量所指向的值。

equals在String中的重写;

public boolean equals(Object anObject) {  
 if (this == anObject) {  
 return true;  
 }  
 if (anObject instanceof String) {  
 String anotherString = (String)anObject;  
 int n = value.length;  
 if (n == anotherString.value.length) {  
 char v1[] = value;  
 char v2[] = anotherString.value;  
 int i = 0;  
 while (n-- != 0) {  
 if (v1[i] != v2[i])  
 return false;  
 i++;  
 }  
 return true;  
 }  
 }  
 return false;  
}

结论:String中的equals比较的是String具体的值。

实际中比较两个对象的值应都使用equals方法不要使用==.

* + 1. **JDK8新特性**

**Stream API**

函数式编程风格。

Stream（流）是一个来自数据源的元素队列并支持聚合操作。

数据源：流的来源， 可以是集合，数组，I/O channel， 产生器generator 等。

聚合操作： 类似SQL语句一样的操作， 比如filter, map, reduce, find, match, sorted等。

具体操作:

1. 生成流

public static void main(String[] args) {  
 List<String> strings = Arrays.*asList*("abc", "", "bc", "efg", "abcd","", "jkl");  
 List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.*toList*());  
 System.*out*.println(filtered);  
}

1. forEach

生成十个随机数并遍历打印

Random random = new Random();  
random.ints().limit(10).forEach(System.*out*::println);

1. map

map 方法用于映射每个元素到对应的结果

List<Integer> numbers = Arrays.*asList*(3, 2, 2, 3, 7, 3, 5,6);  
// 获取对应的平方数  
List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.*toList*());  
System.*out*.println(squaresList);// [9, 4, 49, 25, 36]  
squaresList.stream().forEach(System.*out*::println);

//打印结果为去重后的结果？

1. filter

过滤元素

1. limit

获取指定数量的流

1. sort

排序

Random random = new Random();

// sorted,默认从小到大排序

random.ints().limit(10).sorted().forEach(System.out::println);

1. Collectors

实现规约操作。

List<String>strings = Arrays.*asList*("abc", "", "bc", "efg", "abcd","", "jkl");  
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.*toList*());  
  
System.*out*.println("筛选列表: " + filtered);  
String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.*joining*(", "));  
System.*out*.println("合并字符串: " + mergedString);

1. 统计操作

基础数据类型统计平均值、最大值、最小值时很方便

List<Integer> numbers = Arrays.*asList*(3, 2, 2, 3, 7, 3, 5);  
IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();  
System.*out*.println("列表中最大的数 : " + stats.getMax());  
System.*out*.println("列表中最小的数 : " + stats.getMin());  
System.*out*.println("所有数之和 : " + stats.getSum());  
System.*out*.println("平均数 : " + stats.getAverage());

**lambda表达式**

允许把函数作为一个方法的参数传递到方法中。用于简化实现只有一个未实现方法的接口的匿名使用，如Runnable接口。

用法：

public class LambdaTest {  
 public static void main(String[] args) {  
 new Thread(()->{  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(i);  
 }  
 }).start();  
 }  
}

注意事项：

1. lambda 表达式只能引用标记了 final 的外层局部变量，这就是说不能在 lambda 内部修改定义在域外的局部变量，否则会编译错误。 lambda 表达式中访问外层的局部变量。

2.lambda 表达式的局部变量可以不用声明为 final，但是必须不可被后面的代码修改

public class Java8Tester2 {  
 public static void main(String args[]) {  
 int num = 1; // 隐含默认修饰为final  
 Converter<Integer, String> s = (param) -> System.*out*.println(String.*valueOf*(param + num));  
 s.convert(2); // 输出结果为 3  
 // num = 5; 编译报错，不允许修改  
 }  
  
 public interface Converter<T1, T2> {  
 void convert(int i);  
 }  
}

3. 在 Lambda 表达式当中不允许声明一个与局部变量同名的参数或者局部变量，否则报错。

**默认方法**

一个接口中允许有个实现的方法，该方法使用default修饰。

public interface DefaultMethod {  
 public default void defaultMethod(){  
 System.*out*.println("this is a default method");  
 }  
 void method2();  
}

当一个类实现的多个接口中有相同的默认方法时，有以下几种解决方案：

1. 创建自己的默认方法来覆盖接口中的默认方法。
2. 使用super指定使用哪个接口中的默认方法。

public class DefaultMethod2 implements vehicle1,vehicle2{  
 @Override  
 public void run() {  
 vehicle1.super.run();  
// System.out.println("run in entity");  
 }  
  
 public static void main(String[] args) {  
 new DefaultMethod2().run();  
 }  
  
}  
interface vehicle1{  
 public default void run(){  
 System.*out*.println("run with four wheels");  
 }  
}  
interface vehicle2{  
 public default void run() {  
 System.*out*.println("run with two wheels");  
 }  
}

JDK8中的接口还允许默认静态方法的存在

public class Java8Tester3 implements vehicle3,vehicle4{  
 @Override  
 public void print() {  
 vehicle3.super.print();  
 vehicle4.super.print();  
 vehicle3.*blowHorn*();  
 System.*out*.println("我是一辆汽车!");  
 }  
}  
interface vehicle3{  
 default void print(){  
 System.*out*.println("我是一辆车!");  
 }  
  
 static void blowHorn(){  
 System.*out*.println("按喇叭!!!");  
 }  
 static void blowHorn2(){  
 System.*out*.println("blowhorn2...");  
 }  
}  
interface vehicle4{  
 default void print(){  
 System.*out*.println("我是一辆四轮车!");  
 }  
}

* + 1. **JDK9新特性**

1. **设计模式**
   1. **总览**

设计模式是软件开发中的一套最佳实践，提供了标准的术语系统，具体到特定的场景。通常所说的设计模式是GOF四人帮提出的23种设计模式。GOF的设计模式可以分为三类：

* 创建型模式
* 结构型模式
* 行为型模式

j2ee也提出了几种设计模式。

设计模式中关注的点：

使用场景，解决的问题

设计模式中的角色

使用该设计模式的优缺点

* 1. **创建型设计模式**

**单例模式**

保证一个类仅有一个实例，且提供一个访问它的全局访问点。

使用场景：

需要控制实例数目，节省系统资源时使用单例。

角色：

单例类，私有化构造函数，通过持有自身静态变量并提供公共对外访问该静态变量的接口来保证系统中只存在一个该对象。

优点:减少系统开销，避免资源的多重占用

缺点：与单一职责冲突，需要关注外部对自身的实例化。

**工厂模式**

提供创建对象的最佳方式，不对外暴漏创建逻辑，通过使用一个公共的接口来指向新创建的对象。

使用场景：

复杂对象（一般是由其他对象组合而成）的生成，使用工厂模式比较合适。对于简单对象直接使用new来生成。

角色：

1. 统一的实体模型接口
2. 接口实现类
3. 生成该实体的工厂

优点：调用方便，只需要知道创建对象名称调用对用工厂的生成方法即可；可扩展性强，如果想增加产品，只需要增加产品具体实现即可；调用者不需要关心产品具体实现。

缺点：进行拓展时导致系统中的类成倍增加，增加了系统的复杂性。

**抽象工厂模式**

抽象工厂即生产工厂的工厂。提供创建一系列相关或相互依赖对象的接口，无需指定他们具体实现的类。

使用场景：

系统中的产品多于一个的产品族，而系统只消费其中某一族的产品。

角色：

1. 各种产品实体接口
2. 产品接口实现类
3. 抽象工厂，提供生产各种产品的接口类
4. 具体产品工厂，抽象工厂的具体实现，每种工厂只提供抽象工厂中某一种产品的具体实现。
5. 工厂创建器，通过条件创建指定产品的工厂。

优点：

保证客户端中始终只使用同一个产品族中的对象。

缺点：

增加产品线困难。

**建造者模式**

**原型模式**

* 1. **结构型设计模式**

**适配器模式**

**桥接模式**

**过滤器模式**

**组合模式**

**装饰器模式**

**外观模式**

**享元模式**

**代理模式**

* 1. **行为型设计模式**

**责任链模式**

**命令模式**

**解释器模式**

**迭代器模式**

**中介者模式**

**备忘录模式**

**观察者模式**

**空对象模式**

**策略模式**

**模板模式**

**访问者模式**

* 1. **j2ee设计模式**

**mvc模式**

**业务代理模式**

**组合实体模式**

**数据访问对象模式**

**前端控制器模式**

**拦截过滤器模式**

**服务定位模式**

**传输对象模式**

1. **协议**
   1. **TCP**
   2. **UDP**
   3. **HTTP**
   4. **RPC**
2. **框架**
   1. **mybaits**

工作流程：

1. 创建SqlSessionFactoryBuilder对象并掉要build（）方法创建SqlSessionFactory
2. 通过sqlSessionFactory对象创建SqlSession对象
3. 调用sqlSession的api传入Statement id和参数，最终执行jdbc并执行sql语句，封装结果返回
4. 如果没有直接调用sqlSession的接口，可编写Mapper接口及xml文件（或使用注解），通过sqlSession的getMapper方法拿到mapper对象，执行mapper对象中的接口方法。

核心思想：

使用JDK的动态代理来实现Mapper的实例化并执行相应的方法，实际通过InvocationHandler的invoke方法来完成sql的解析与执行。

resultType

只能做到表字段到java对象属性的一一对应，且字段名必须一致

resultMap

可以指定表字段到java对象属性的对应关系

resultMap中能使用的属性

association： 将关联查询信息映射到一个pojo对象中。

collection：将关联查询信息映射到一个list集合中。

mybatis-spring

两个重要的组件

MapperScannerConfigurer，负责扫描指定包下的映射接口并向容器中注册对应的bean

SqlSessionFactoryBean，一个FactoryBean，负责创建SqlSessionFactory，而SqlSessionFactory是创建SqlSession的工厂类。

* 1. **spring**

启动过程

上下监听器ContextLoaderListener的启动

前端控制器DispatcherServlet的启动

* 1. **spring mvc**

springmvc处理请求流程

1.请求发送到DispatcherServlet，DispatcherServlet通过Handlermapping找到具体的拦截器、处理器，DispatcherServlet再调用HandlerAdapter处理器适配器找到具体的处理器，处理器执行完毕后返回MV（ModelAndView），HandlerAdapter将执行结果mv返回给DispatcherServlet，DispatcherServlet将mv传给ViewReslover，viewReslover返回具体的View，DispatcherServlet根据view进行视图渲染响应给用户。

spring 与spring mvc注解配置的区别

一般都是在spring 中配置注解扫描dao、service层，而在springMVC中配置注解扫描controller。

* 1. **spring boot**
  2. **spring cloud**
  3. **dubbo**
  4. **k8s**

1. **微服务架构**
   1. **概念**
   2. **微服务基本组件**
   3. **Spring cloud netflix**
   4. **Spring cloud**
2. **消息中间件**
3. **持续集成持续交付**
   1. **一些概念**

管道：将源代码转换为可以发布产品的多个不同的任务（task）和作业（job）通常串联成为一个软件“管道”，一个自动化流程成功后会启动管道中的下一个流程。管道的工作之一就是快速处理变更。

快速失败（fail fast）：快速失败值的是在管道流程中尽快发现问题并快速通知用户的方式，这样可以及时修正问题并提交代码以便使管道再次运行。

* 1. **持续集成（continuous interagtion，CI）**

持续集成是在源代码变更后自动检测、拉取、构建和进行单元测试的过程。

持续集成基本思想：让一个自动化程序监测一个或多个源代码仓库是否又变更，当变更推送到仓库时，它会监测到变更、下载副本、构建并进行单元测试。持续集成中会使用jenkins这样的工具。

* 1. **持续交付（continuous Derlivery， CD）**

持续交付通常是指整个流程链，他自动检测源代码变更并通过构建、测试、打包和相关操作运行以生成可部署的版本，基本上不需要任何人为干预。

* 1. **maven**
  2. **jekins**
  3. **docker**