1. 一个类永远不要去继承一个已经实现好的类。而只能继承抽象类或实现接口
2. 1、instanceof用于判断一个对象是否是某个类的实例。
3. 2、在对象向下转型之前最好使用instanceof关键字进行验证。

**自动绑定**

**@autowire自动注解**

**把把类实例化成bean加载进来，**

**\*注意，没有自定义构造函数**

**\*自动注入的bean是单例的，即整个系统中只要是通过自动注入的同一个类，都是一个对象，优点：不用每次创建对象，对象状态不需改变，大多数用map集合，**

**\*用component注解的类，已经把类初始一个实例A1并装入容器，程序中用@autowire加载的bean就是A1,hashcode一样**

**1 .泛型**

**a.泛型类**

**定义时可指明类型只能为类，更安全，不会在编译运行的时候才报错， 例如list<integer>**

**List.add(“ddd”)就会报错**

**b.泛型方法（提高重用性）**

对java的泛型特性的了解仅限于表面的浅浅一层，直到在学习设计模式时发现有不了解的用法，才想起详细的记录一下。

本文参考java 泛型详解、Java中的泛型方法、 java泛型详解

# 1. 概述

泛型在java中有很重要的地位，在面向对象编程及各种设计模式中有非常广泛的应用。

什么是泛型？为什么要使用泛型？

[复制代码](javascript:void(0);)

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？  
  
顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），  
  
然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，  
  
操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

[复制代码](javascript:void(0);)

# 22多态

**（重写父类方法时可以不加overvide,但写错名字不会报错，最好写上）**

父类引用子类对象时，可以调用重写过父类方法的子类方法。即cat1.eat调用cat类的方法，但不可以调用 子类的单独的drink方法，

\*\*即对象还是父类对象，只能调用父类有的方法，2中情况

1. 重写了父类的方法，那么就实际调用的子类
2. 没有重写父类的，就是实际调用父类的方法
3. 其实还是父类对象，不能调用子类中有而父类没有的方法

总结：一般父类用抽象类表示，子类直接用实际类，一般情况下父类的方法能满足子类的使用时，直接父类引用子类，用工厂模式构建（泛型类），当父类满足不了时，需要特殊处理时，重写父类方法，那么父类引用子类时就调用的是子类的方法

工厂模式：

例如：

Msgfactory类，子类为queryRequestMsg.class

父类构造器方法

public <T extends AbstractMsg> T newInstance(Class<T> clz) {  
 try {  
 return clz.newInstance();  
 } catch (Exception e) {  
 *log*.error("new instance for {} failed! cause:{}", clz, JSONUtil.*objToJson*(e));  
 }  
 return null;  
}

例1

Class animal{

Public viod eat()

Sout(animal eat)

}

Class cat extends animal{

@overvide

Public viod eat()

Sout(cat eat)

}

Public viod drink()

Sout(cat drink)

}

}

Animal cat1=new cat();

# 22抽象类 abstract

\*子类继承抽象类时，必须实现父类的抽象方法（空的）

例1：**protected abstract void** channelRead0(ChannelHandlerContext var1, I var2) **throws** Exception;

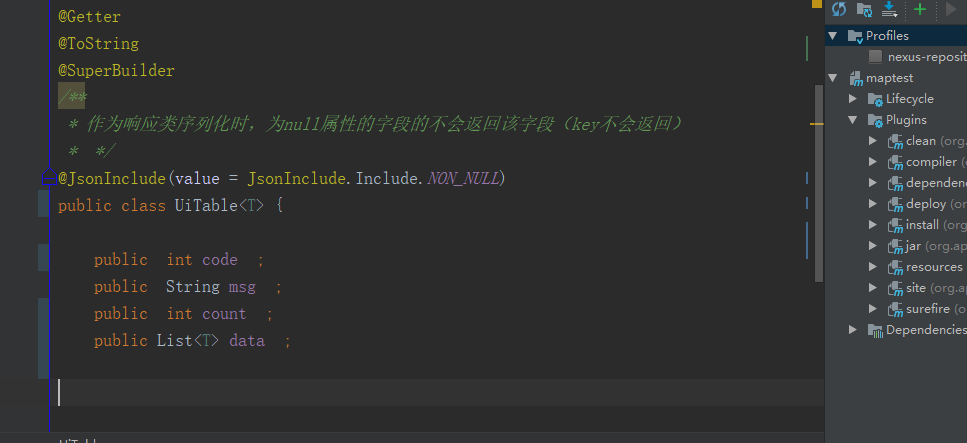
\*&所有类型的父类如果有有参的构造函数，子类必须定义构造有参构造函数

Class(P.a,P.b){

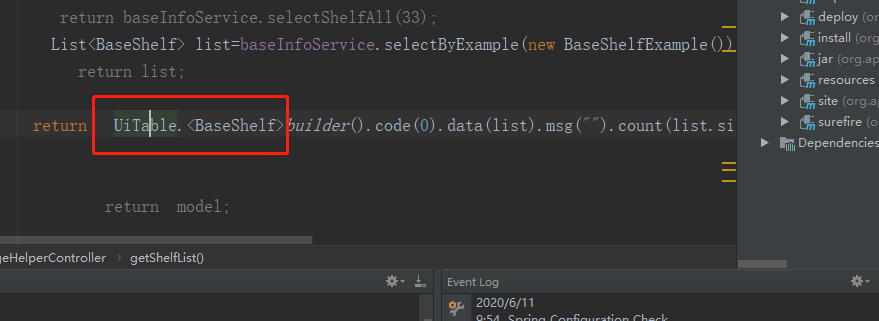
Super(a,b)}

# 24.泛型类的构造

定义泛型类



构造器构造Clase.<T>.builder



26.数据库分层

Dao层，操作数据库的原始接口层，增删改查，直接调用就可以返回定义的类型

Service层，具体怎么操作数据库的实现层，可能查询的就是某一列；

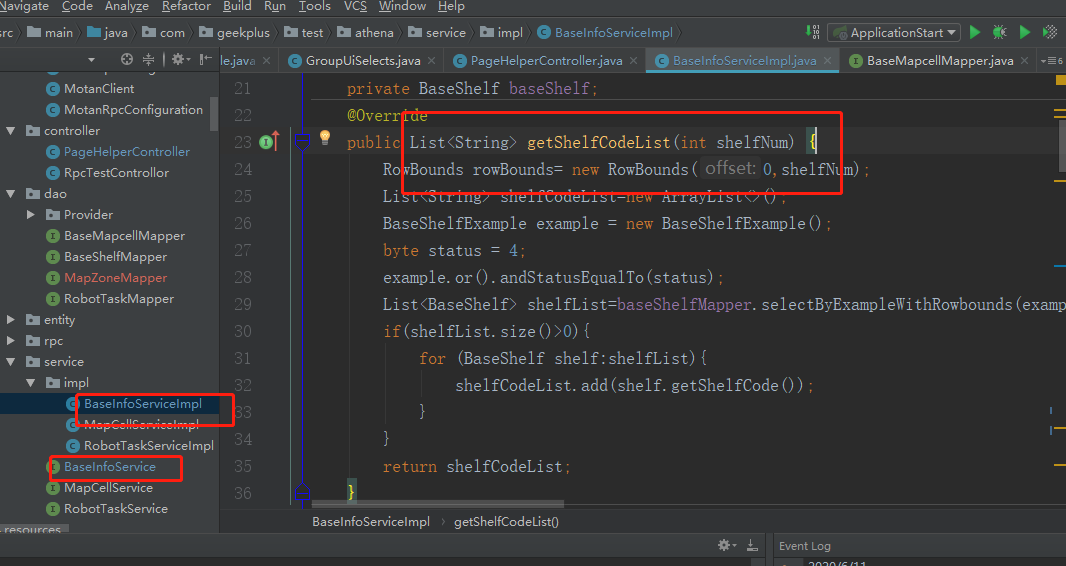
1分为定义接口定义查询结果的类型

2.实现具体操作，这里边直接调用的是dao层接口

Case接口



Case2具体操作daoceng



# 25.多线程对象锁synchronized

## 25.1对象锁

**private** **synchronized void** callback() {  
 **if** (CollUtil.*isEmpty*(**clientCallbackMap**)) {  
 **return**;  
 }  
 **for** (Entry<String, Queue<CallbackEntry>> ent : **clientCallbackMap**.entrySet()) {  
 Queue<CallbackEntry> channelQueue = ent.getValue();  
 **try** {  
 **for** (CallbackEntry entry; (entry = channelQueue.peek()) != **null**; ) {  
 **boolean** isSuccess = callback(entry);  
 **if** (isSuccess) {  
 channelQueue.remove(entry);  
 ***log***.info(**"callback completed. msg:{}, sendTimes:{}"**, entry.getMsg(), entry.getSendTimes());  
 } **else** {  
 *// 失败则调出循环，发送下一个channel的回调* **break**;  
 }  
 }  
 } **catch** (Exception e) {  
 ***log***.error(**"caught an error when callback. channelId:{}, msg:{}, exception:{}"**,  
 ent.getKey(), JSONUtil.*objToJson*(channelQueue.peek()), ExceptionUtil.*getStackTrace*(e));  
 }  
 }  
  
}

**知识点： 1**当synchronized修饰多线程调用方法时callback()，当有一个线程A调用callback()时，其他线程必须等待A 线程调用完毕再调用

2 当定义的方法返回值为空时，满足对象不为空才执行下面的代码，否则直接反回空；

对象锁知识：

例1：**if** (CollUtil.*isEmpty*(**clientCallbackMap**)) {  
 **return**;  
 }

一个被举了无数次的例子：

[复制代码](javascript:void(0);)

List arrayList = new ArrayList();

arrayList.add("aaaa");

arrayList.add(100);

for(int i = 0; i< arrayList.size();i++){

String item = (String)arrayList.get(i);

Log.d("泛型测试","item = " + item);

}

[复制代码](javascript:void(0);)

毫无疑问，程序的运行结果会以崩溃结束：

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

ArrayList可以存放任意类型，例子中添加了一个String类型，添加了一个Integer类型，再使用时都以String的方式使用，因此程序崩溃了。为了解决类似这样的问题（在编译阶段就可以解决），泛型应运而生。

我们将第一行声明初始化list的代码更改一下，编译器会在编译阶段就能够帮我们发现类似这样的问题。

List<String> arrayList = new ArrayList<String>();

...

//arrayList.add(100); 在编译阶段，编译器就会报错

多线程锁的详解：

1.为什么要使用synchronized

在并发编程中存在线程安全问题，主要原因有：1.存在共享数据 2.多线程共同操作共享数据。关键字synchronized可以保证在同一时刻，只有一个线程可以执行某个方法或某个代码块，同时synchronized可以保证一个线程的变化可见（可见性），即可以代替volatile。

2.实现原理

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

3.synchronized的三种应用方式

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

普通同步方法（实例方法），锁是当前实例对象 ，进入同步代码前要获得当前实例的锁

静态同步方法，锁是当前类的class对象 ，进入同步代码前要获得当前类对象的锁

同步方法块，锁是括号里面的对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

4.synchronized的作用

Synchronized是Java中解决并发问题的一种最常用最简单的方法 ，他可以确保线程互斥的访问同步代码

5.举栗子

\*\*一、synchronized作用于实例方法\*\*

①多个线程访问同一个对象的同一个方法

public class synchronizedTest implements Runnable {

//共享资源

static int i =0;

/\*\*

\* synchronized 修饰实例方法

\*/

public synchronized void increase(){

i++;

}

@Override

public void run(){

for (int j =0 ; j<10000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

synchronizedTest test = new synchronizedTest();

Thread t1 = new Thread(test);

Thread t2 = new Thread(test);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println(i);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

结果：

分析：当两个线程同时对一个对象的一个方法进行操作，只有一个线程能够抢到锁。因为一个对象只有一把锁，一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，就不能访问该对象的其他synchronized实例方法，但是可以访问非synchronized修饰的方法

②一个线程获取了该对象的锁之后，其他线程来访问其他synchronized实例方法现象 举栗

public class SynchronizedTest {

public synchronized void method1() {

System.out.println("Method 1 start");

try {

System.out.println("Method 1 execute");

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Method 1 end");

}

public synchronized void method2() {

System.out.println("Method 2 start");

try {

System.out.println("Method 2 execute");

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Method 2 end");

}

public static void main(String[] args) {

final SynchronizedTest test = new SynchronizedTest();

new Thread(test::method1).start();

new Thread(test::method2).start();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

结果：

分析：可以看出其他线程来访问synchronized修饰的其他方法时需要等待线程1先把锁释放

③一个线程获取了该对象的锁之后，其他线程来访问其他非synchronized实例方法现象 举栗

去掉②中方法二的synchronized

public class SynchronizedTest {

public synchronized void method1() {

System.out.println("Method 1 start");

try {

System.out.println("Method 1 execute");

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Method 1 end");

}

public void method2() {

System.out.println("Method 2 start");

try {

System.out.println("Method 2 execute");

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Method 2 end");

}

public static void main(String[] args) {

final SynchronizedTest test = new SynchronizedTest();

new Thread(test::method1).start();

new Thread(test::method2).start();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

结果：

分析：当线程1还在执行时，线程2也执行了，所以当其他线程来访问非synchronized修饰的方法时是可以访问的

④当多个线程作用于不同的对象

public class SynchronizedTest {

public synchronized void method1() {

System.out.println("Method 1 start");

try {

System.out.println("Method 1 execute");

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Method 1 end");

}

public synchronized void method2() {

System.out.println("Method 2 start");

try {

System.out.println("Method 2 execute");

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Method 2 end");

}

public static void main(String[] args) {

final SynchronizedTest test1 = new SynchronizedTest();

final SynchronizedTest test2 = new SynchronizedTest();

new Thread(test1::method1).start();

new Thread(test2::method2).start();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

结果：

分析：因为两个线程作用于不同的对象，获得的是不同的锁，所以互相并不影响

\*\*此处思考一个问题：为什么分布式环境下synchronized失效？如何解决这种情况？\*\*

\*\*二、synchronized作用于静态方法\*\*

public class synchronizedTest implements Runnable {

//共享资源

static int i =0;

/\*\*

\* synchronized 修饰实例方法

\*/

public static synchronized void increase(){

i++;

}

@Override

public void run(){

for (int j =0 ; j<10000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

Thread t1 = new Thread(new synchronizedTest());

Thread t2 = new Thread(new synchronizedTest());

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println(i);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

结果：

分析：由例子可知，两个线程实例化两个不同的对象，但是访问的方法是静态的，两个线程发生了互斥（即一个线程访问，另一个线程只能等着），因为静态方法是依附于类而不是对象的，当synchronized修饰静态方法时，锁是class对象。

\*\*三、synchronized作用于同步代码块\*\*

为什么要同步代码块呢？在某些情况下，我们编写的方法体可能比较大，同时存在一些比较耗时的操作，而需要同步的代码又只有一小部分，如果直接对整个方法进行同步操作，可能会得不偿失，此时我们可以使用同步代码块的方式对需要同步的代码进行包裹，这样就无需对整个方法进行同步操作了。

public class synchronizedTest implements Runnable {

static synchronizedTest instance=new synchronizedTest();

static int i=0;

@Override

public void run() {

//省略其他耗时操作....

//使用同步代码块对变量i进行同步操作,锁对象为instance

synchronized(instance){

for(int j=0;j<10000;j++){

i++;

}

}

}

public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println(i);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

结果：

分析：将synchronized作用于一个给定的实例对象instance，即当前实例对象就是锁对象，每次当线程进入synchronized包裹的代码块时就会要求当前线程持有instance实例对象锁，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待，这样也就保证了每次只有一个线程执行i++;操作。当然除了instance作为对象外，我们还可以使用this对象(代表当前实例)或者当前类的class对象作为锁，如下代码：

//this,当前实例对象锁

synchronized(this){

for(int j=0;j<1000000;j++){

i++;

}

}

//class对象锁

synchronized(AccountingSync.class){

for(int j=0;j<1000000;j++){

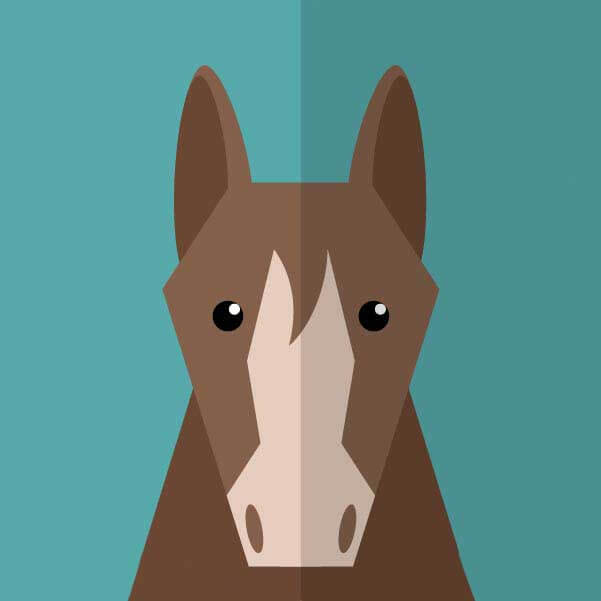
i++;

}

}

## 25.2多线程并发

### 1. countDownLatch

[](https://www.jianshu.com/u/174bbbbc0679)

[指尖架构141319](https://www.jianshu.com/u/174bbbbc0679)关注

12019.03.01 11:16:02字数 295阅读 79,854

#### 1.背景：

* countDownLatch是在java1.5被引入，跟它一起被引入的工具类还有CyclicBarrier、Semaphore、concurrentHashMap和BlockingQueue。
* 存在于java.util.cucurrent包下。

#### 2.概念

* countDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行。
* 是通过一个计数器来实现的，计数器的初始值是线程的数量。每当一个线程执行完毕后，计数器的值就-1，当计数器的值为0时，表示所有线程都执行完毕，然后在闭锁上等待的线程就可以恢复工作了。

#### 3.源码

* countDownLatch类中只提供了一个构造器：

//参数count为计数值

public CountDownLatch(int count) { };

* 类中有三个方法是最重要的：

//调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

public void await() throws InterruptedException { };

//和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };

//将count值减1

public void countDown() { };

#### 4.示例

普通示例：

public class CountDownLatchTest {

public static void main(String[] args) {

final CountDownLatch latch = new CountDownLatch(2);

System.out.println("主线程开始执行…… ……");

//第一个子线程执行

ExecutorService es1 = Executors.newSingleThreadExecutor();

es1.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(3000);

System.out.println("子线程："+Thread.currentThread().getName()+"执行");

} catch (InterruptedException e) {

e.printStackTrace();

}

latch.countDown();

}

});

es1.shutdown();

//第二个子线程执行

ExecutorService es2 = Executors.newSingleThreadExecutor();

es2.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("子线程："+Thread.currentThread().getName()+"执行");

latch.countDown();

}

});

es2.shutdown();

System.out.println("等待两个线程执行完毕…… ……");

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("两个子线程都执行完毕，继续执行主线程");

}

}

结果集：

主线程开始执行…… ……

等待两个线程执行完毕…… ……

子线程：pool-1-thread-1执行

子线程：pool-2-thread-1执行

两个子线程都执行完毕，继续执行主线程

模拟并发示例：

public class Parallellimit {

public static void main(String[] args) {

ExecutorService pool = Executors.newCachedThreadPool();

CountDownLatch cdl = new CountDownLatch(100);

for (int i = 0; i < 100; i++) {

CountRunnable runnable = new CountRunnable(cdl);

pool.execute(runnable);

}

}

}

class CountRunnable implements Runnable {

private CountDownLatch countDownLatch;

public CountRunnable(CountDownLatch countDownLatch) {

this.countDownLatch = countDownLatch;

}

@Override

public void run() {

try {

synchronized (countDownLatch) {

/\*\*\* 每次减少一个容量\*/

countDownLatch.countDown();

System.out.println("thread counts = " + (countDownLatch.getCount()));

}

countDownLatch.await();

System.out.println("concurrency counts = " + (100 - countDownLatch.getCount()));

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

\***CountDownLatch和CyclicBarrier区别：**  
1.countDownLatch是一个计数器，线程完成一个记录一个，计数器递减，只能只用一次  
2.CyclicBarrier的计数器更像一个阀门，需要所有线程都到达，然后继续执行，计数器递增，提供reset功能，可以多次使用

### 2.高并发代码

## 1、并发程序

/\*\*

\* 高并发测试：

\* 创建threadNum个线程；

\* 执行任务task

\* @param threadNum 线程数量

\* @param task 任务

\*/

public static void parallelTesk(int threadNum, Runnable task){

// 1. 定义闭锁来拦截线程

final CountDownLatch startGate = new CountDownLatch(1);

final CountDownLatch endGate = new CountDownLatch(threadNum);

// 2. 创建指定数量的线程

for (int i = 0; i <threadNum; i++) {

Thread t = new Thread(() -> {

try {

startGate.await();

try {

task.run();

} finally {

endGate.countDown();

}

} catch (InterruptedException e) {

}

});

t.start();

}

// 3. 线程统一放行，并记录时间！

long start = System.nanoTime();

startGate.countDown();

try {

endGate.await();

} catch (InterruptedException e) {

}

long end = System.nanoTime();

System.out.println("cost times :" +(end - start));

}

## 2、应用

import java.util.concurrent.CountDownLatch;

public class Main {

/\*\*

\* 定义一个不安全方法：在高并发情况下得不到想要的结果

\* queryTimes 是一个不安全的属性；

\* getTimes是一个不安全方法；

\*/

static int queryTimes = 0;

public static int getTimes(){

queryTimes = queryTimes +1;

return queryTimes;

}

/\*\*

\* 200个线程来执行获取getTimes

\* @param args

\*/

public static void main(String[] args) {

parallelTesk(200, new Runnable() {

@Override

public void run() {

System.out.println(getTimes());

}

});

}

/\*\*

\* 高并发测试：

\* 创建threadNum个线程；

\* 执行任务task

\* @param threadNum 线程数量

\* @param task 任务

\*/

public static void parallelTesk(int threadNum, Runnable task){

// 1. 定义闭锁来拦截线程

final CountDownLatch startGate = new CountDownLatch(1);

final CountDownLatch endGate = new CountDownLatch(threadNum);

// 2. 创建指定数量的线程

for (int i = 0; i <threadNum; i++) {

Thread t = new Thread(() -> {

try {

startGate.await();

try {

task.run();

} finally {

endGate.countDown();

}

} catch (InterruptedException e) {

}

});

t.start();

}

// 3. 线程统一放行，并记录时间！

long start = System.nanoTime();

startGate.countDown();

try {

endGate.await();

} catch (InterruptedException e) {

}

long end = System.nanoTime();

System.out.println("cost times :" +(end - start));

}

}

# 3.泛型 特性

泛型只在编译阶段有效。看下面的代码：

[复制代码](javascript:void(0);)

List<String> stringArrayList = new ArrayList<String>();

List<Integer> integerArrayList = new ArrayList<Integer>();

Class classStringArrayList = stringArrayList.getClass();

Class classIntegerArrayList = integerArrayList.getClass();

if(classStringArrayList.equals(classIntegerArrayList)){

Log.d("泛型测试","类型相同");

}

[复制代码](javascript:void(0);)

输出结果：D/泛型测试: 类型相同。

通过上面的例子可以证明，在编译之后程序会采取去泛型化的措施。也就是说Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

# 4. 泛型的使用

泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法

## 4.3 泛型类

泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。

泛型类的最基本写法（这么看可能会有点晕，会在下面的例子中详解）：

class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{

private 泛型标识 /\*（成员变量类型）\*/ var;

.....

}

}

一个最普通的泛型类：

[复制代码](javascript:void(0);)

//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型

//在实例化泛型类时，必须指定T的具体类型

public class Generic<T>{

//key这个成员变量的类型为T,T的类型由外部指定

private T key;

public Generic(T key) { //泛型构造方法形参key的类型也为T，T的类型由外部指定

this.key = key;

}

public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定

return key;

}

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//泛型的类型参数只能是类类型（包括自定义类），不能是简单类型

//传入的实参类型需与泛型的类型参数类型相同，即为Integer.

Generic<Integer> genericInteger = new Generic<Integer>(123456);

//传入的实参类型需与泛型的类型参数类型相同，即为String.

Generic<String> genericString = new Generic<String>("key\_vlaue");

Log.d("泛型测试","key is " + genericInteger.getKey());

Log.d("泛型测试","key is " + genericString.getKey());

[复制代码](javascript:void(0);)

12-27 09:20:04.432 13063-13063/? D/泛型测试: key is 123456

12-27 09:20:04.432 13063-13063/? D/泛型测试: key is key\_vlaue

定义的泛型类，就一定要传入泛型类型实参么？并不是这样，在使用泛型的时候如果传入泛型实参，则会根据传入的泛型实参做相应的限制，此时泛型才会起到本应起到的限制作用。如果不传入泛型类型实参的话，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。

看一个例子：

[复制代码](javascript:void(0);)

Generic generic = new Generic("111111");

Generic generic1 = new Generic(4444);

Generic generic2 = new Generic(55.55);

Generic generic3 = new Generic(false);

Log.d("泛型测试","key is " + generic.getKey());

Log.d("泛型测试","key is " + generic1.getKey());

Log.d("泛型测试","key is " + generic2.getKey());

Log.d("泛型测试","key is " + generic3.getKey());

[复制代码](javascript:void(0);)

D/泛型测试: key is 111111

D/泛型测试: key is 4444

D/泛型测试: key is 55.55

D/泛型测试: key is false

**注意：**

* 泛型的类型参数只能是类类型，不能是简单类型。
* 不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。

　　if(ex\_num instanceof Generic<Number>){ }

## 4.4 泛型接口

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中，可以看一个例子：

//定义一个泛型接口

public interface Generator<T> {

public T next();

}

当实现泛型接口的类，未传入泛型实参时：

[复制代码](javascript:void(0);)

/\*\*

\* 未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加到类中

\* 即：class FruitGenerator<T> implements Generator<T>{

\* 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"

\*/

class FruitGenerator<T> implements Generator<T>{

@Override

public T next() {

return null;

}

}

[复制代码](javascript:void(0);)

当实现泛型接口的类，传入泛型实参时：

[复制代码](javascript:void(0);)

/\*\*

\* 传入泛型实参时：

\* 定义一个生产器实现这个接口,虽然我们只创建了一个泛型接口Generator<T>

\* 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。

\* 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要替换成传入的实参类型

\* 即：Generator<T>，public T next();中的的T都要替换成传入的String类型。

\*/

public class FruitGenerator implements Generator<String> {

private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

@Override

public String next() {

Random rand = new Random();

return fruits[rand.nextInt(3)];

}

}

[复制代码](javascript:void(0);)

## 4.5 泛型通配符

我们知道Ingeter是Number的一个子类，同时在特性章节中我们也验证过Generic<Ingeter>与Generic<Number>实际上是相同的一种基本类型。那么问题来了，在使用Generic<Number>作为形参的方法中，能否使用Generic<Ingeter>的实例传入呢？在逻辑上类似于Generic<Number>和Generic<Ingeter>是否可以看成具有父子关系的泛型类型呢？

为了弄清楚这个问题，我们使用Generic<T>这个泛型类继续看下面的例子：

public void showKeyValue1(Generic<Number> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

[复制代码](javascript:void(0);)

Generic<Integer> gInteger = new Generic<Integer>(123);

Generic<Number> gNumber = new Generic<Number>(456);

showKeyValue(gNumber);

// showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer>

// cannot be applied to Generic<java.lang.Number>

// showKeyValue(gInteger);

[复制代码](javascript:void(0);)

通过提示信息我们可以看到Generic<Integer>不能被看作为`Generic<Number>的子类。由此可以看出:同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。

回到上面的例子，如何解决上面的问题？总不能为了定义一个新的方法来处理Generic<Integer>类型的类，这显然与java中的多台理念相违背。因此我们需要一个在逻辑上可以表示同时是Generic<Integer>和Generic<Number>父类的引用类型。由此类型通配符应运而生。

我们可以将上面的方法改一下：

public void showKeyValue1(Generic<?> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

类型通配符一般是使用？代替具体的类型实参，注意了，此处’？’是类型实参，而不是类型形参 。重要说三遍！此处’？’是类型实参，而不是类型形参 ！ 此处’？’是类型实参，而不是类型形参 ！再直白点的意思就是，此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型的父类。是一种真实的类型。

可以解决当具体类型不确定的时候，这个通配符就是 ?  ；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用 ? 通配符来表未知类型。

**4.6 泛型方法**

在java中,泛型类的定义非常简单，但是泛型方法就比较复杂了。

尤其是我们见到的大多数泛型类中的成员方法也都使用了泛型，有的甚至泛型类中也包含着泛型方法，这样在初学者中非常容易将泛型方法理解错了。

泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型 。

[复制代码](javascript:void(0);)

/\*\*

\* 泛型方法的基本介绍

\* @param tClass 传入的泛型实参

\* @return T 返回值为T类型

\* 说明：

\* 1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。

\* 2）只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。

\* 3）<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。

\* 4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。

\*/

public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,

IllegalAccessException{

T instance = tClass.newInstance();

return instance;

}

[复制代码](javascript:void(0);)

Object obj = genericMethod(Class.forName("com.test.test"));

**4.6.1 泛型方法的基本用法**

光看上面的例子有的同学可能依然会非常迷糊，我们再通过一个例子，把我泛型方法再总结一下。

[复制代码](javascript:void(0);)

public class GenericTest {

//这个类是个泛型类，在上面已经介绍过

public class Generic<T>{

private T key;

public Generic(T key) {

this.key = key;

}

//我想说的其实是这个，虽然在方法中使用了泛型，但是这并不是一个泛型方法。

//这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明过的泛型。

//所以在这个方法中才可以继续使用 T 这个泛型。

public T getKey(){

return key;

}

/\*\*

\* 这个方法显然是有问题的，在编译器会给我们提示这样的错误信息"cannot reslove symbol E"

\* 因为在类的声明中并未声明泛型E，所以在使用E做形参和返回值类型时，编译器会无法识别。

public E setKey(E key){

this.key = keu

}

\*/

}

/\*\*

\* 这才是一个真正的泛型方法。

\* 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛型T

\* 这个T可以出现在这个泛型方法的任意位置.

\* 泛型的数量也可以为任意多个

\* 如：public <T,K> K showKeyName(Generic<T> container){

\* ...

\* }

\*/

public <T> T showKeyName(Generic<T> container){

System.out.println("container key :" + container.getKey());

//当然这个例子举的不太合适，只是为了说明泛型方法的特性。

T test = container.getKey();

return test;

}

//这也不是一个泛型方法，这就是一个普通的方法，只是使用了Generic<Number>这个泛型类做形参而已。

public void showKeyValue1(Generic<Number> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

//这也不是一个泛型方法，这也是一个普通的方法，只不过使用了泛型通配符?

//同时这也印证了泛型通配符章节所描述的，?是一种类型实参，可以看做为Number等所有类的父类

public void showKeyValue2(Generic<?> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

/\*\*

\* 这个方法是有问题的，编译器会为我们提示错误信息："UnKnown class 'E' "

\* 虽然我们声明了<T>,也表明了这是一个可以处理泛型的类型的泛型方法。

\* 但是只声明了泛型类型T，并未声明泛型类型E，因此编译器并不知道该如何处理E这个类型。

public <T> T showKeyName(Generic<E> container){

...

}

\*/

/\*\*

\* 这个方法也是有问题的，编译器会为我们提示错误信息："UnKnown class 'T' "

\* 对于编译器来说T这个类型并未项目中声明过，因此编译也不知道该如何编译这个类。

\* 所以这也不是一个正确的泛型方法声明。

public void showkey(T genericObj){

}

\*/

public static void main(String[] args) {

}

}

[复制代码](javascript:void(0);)

**4.6.2 类中的泛型方法**

当然这并不是泛型方法的全部，泛型方法可以出现杂任何地方和任何场景中使用。但是有一种情况是非常特殊的，当泛型方法出现在泛型类中时，我们再通过一个例子看一下

[复制代码](javascript:void(0);)

public class GenericFruit {

class Fruit{

@Override

public String toString() {

return "fruit";

}

}

class Apple extends Fruit{

@Override

public String toString() {

return "apple";

}

}

class Person{

@Override

public String toString() {

return "Person";

}

}

class GenerateTest<T>{

public void show\_1(T t){

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

}

//在泛型类中声明了一个泛型方法，使用泛型E，这种泛型E可以为任意类型。可以类型与T相同，也可以不同。

//由于泛型方法在声明的时候会声明泛型<E>，因此即使在泛型类中并未声明泛型，编译器也能够正确识别泛型方法中识别的泛型。

public <E> void show\_3(E t){

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

}

//在泛型类中声明了一个泛型方法，使用泛型T，注意这个T是一种全新的类型，可以与泛型类中声明的T不是同一种类型。

public <T> void show\_2(T t){

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

}

}

public static void main(String[] args) {

Apple apple = new Apple();

Person person = new Person();

GenerateTest<Fruit> generateTest = new GenerateTest<Fruit>();

//apple是Fruit的子类，所以这里可以

generateTest.show\_1(apple);

//编译器会报错，因为泛型类型实参指定的是Fruit，而传入的实参类是Person

//generateTest.show\_1(person);

//使用这两个方法都可以成功

generateTest.show\_2(apple);

generateTest.show\_2(person);

//使用这两个方法也都可以成功

generateTest.show\_3(apple);

generateTest.show\_3(person);

}

}

[复制代码](javascript:void(0);)

**4.6.3 泛型方法与可变参数**

再看一个泛型方法和可变参数的例子：

public <T> void printMsg( T... args){

for(T t : args){

Log.d("泛型测试","t is " + t);

}

}

printMsg("111",222,"aaaa","2323.4",55.55);

**4.6.4 静态方法与泛型**

静态方法有一种情况需要注意一下，那就是在类中的静态方法使用泛型：静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

即：如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法 。

[复制代码](javascript:void(0);)

public class StaticGenerator<T> {

....

....

/\*\*

\* 如果在类中定义使用泛型的静态方法，需要添加额外的泛型声明（将这个方法定义成泛型方法）

\* 即使静态方法要使用泛型类中已经声明过的泛型也不可以。

\* 如：public static void show(T t){..},此时编译器会提示错误信息：

"StaticGenerator cannot be refrenced from static context"

\*/

public static <T> void show(T t){

}

}

[复制代码](javascript:void(0);)

**4.6.5 泛型方法总结**

泛型方法能使方法独立于类而产生变化，以下是一个基本的指导原则：

无论何时，如果你能做到，你就该尽量使用泛型方法。也就是说，如果使用泛型方法将整个类泛型化，  
  
那么就应该使用泛型方法。另外对于一个static的方法而已，无法访问泛型类型的参数。  
  
所以如果static方法要使用泛型能力，就必须使其成为泛型方法。

**4.6 泛型上下边界**

在使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

public void showKeyValue1(Generic<? extends Number> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

[复制代码](javascript:void(0);)

Generic<String> generic1 = new Generic<String>("11111");

Generic<Integer> generic2 = new Generic<Integer>(2222);

Generic<Float> generic3 = new Generic<Float>(2.4f);

Generic<Double> generic4 = new Generic<Double>(2.56);

//这一行代码编译器会提示错误，因为String类型并不是Number类型的子类

//showKeyValue1(generic1);

showKeyValue1(generic2);

showKeyValue1(generic3);

showKeyValue1(generic4);

[复制代码](javascript:void(0);)

如果我们把泛型类的定义也改一下:

[复制代码](javascript:void(0);)

public class Generic<T extends Number>{

private T key;

public Generic(T key) {

this.key = key;

}

public T getKey(){

return key;

}

}

[复制代码](javascript:void(0);)

//这一行代码也会报错，因为String不是Number的子类

Generic<String> generic1 = new Generic<String>("11111");

再来一个泛型方法的例子：

[复制代码](javascript:void(0);)

//在泛型方法中添加上下边界限制的时候，必须在权限声明与返回值之间的<T>上添加上下边界，即在泛型声明的时候添加

//public <T> T showKeyName(Generic<T extends Number> container)，编译器会报错："Unexpected bound"

public <T extends Number> T showKeyName(Generic<T> container){

System.out.println("container key :" + container.getKey());

T test = container.getKey();

return test;

}

[复制代码](javascript:void(0);)

通过上面的两个例子可以看出：泛型的上下边界添加，必须与泛型的声明在一起 。

**4.7 关于泛型数组要提一下**

看到了很多文章中都会提起泛型数组，经过查看sun的说明文档，在java中是”不能创建一个确切的泛型类型的数组”的。

也就是说下面的这个例子是不可以的：

List<String>[] ls = new ArrayList<String>[10];

而使用通配符创建泛型数组是可以的，如下面这个例子：

List<?>[] ls = new ArrayList<?>[10];

这样也是可以的：

List<String>[] ls = new ArrayList[10];

下面使用[Sun的一篇文档](http://docs.oracle.com/javase/tutorial/extra/generics/fineprint.html)的一个例子来说明这个问题：

[复制代码](javascript:void(0);)

List<String>[] lsa = new List<String>[10]; // Not really allowed.

Object o = lsa;

Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();

li.add(new Integer(3));

oa[1] = li; // Unsound, but passes run time store check

String s = lsa[1].get(0); // Run-time error: ClassCastException.

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList而不会出现异常，  
  
但是在取出数据的时候却要做一次类型转换，所以就会出现ClassCastException，如果可以进行泛型数组的声明，  
  
上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。

而对泛型数组的声明进行限制，对于这样的情况，可以在编译期提示代码有类型安全问题，比没有任何提示要强很多。

[复制代码](javascript:void(0);)

下面采用通配符的方式是被允许的:数组的类型不可以是类型变量，除非是采用通配符的方式，因为对于通配符的方式，最后取出数据是要做显式的类型转换的。

[复制代码](javascript:void(0);)

List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.

Object o = lsa;

Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();

li.add(new Integer(3));

oa[1] = li; // Correct.

Integer i = (Integer) lsa[1].get(0); // OK

# 5.0.泛型接口及接口继承接口

**（接口可继承多个接口，但类只能继承一个类）的使用**

**1．泛型接口：更抽象化，为了复用性更强，接口涉及形参各种各样时用泛型接口；如下入参可能是user,student：**

**public interface BaseMapper<T> {  
　　public T selectByPrimaryKey(Integer id);  
　　public void insert(T t);  
　　public void updateByPrimaryKey(Integer id);  
　　public void deleteByPrimaryKey(Integer id);  
}**

**\*子接口继承父接口：公共的接口在父接口，子接口还可以扩充自己的接口，但子接口的实现类必须实现所有的接口方法；**

**public interface UserMapper extends BaseMapper<User>{  
　　　　public int selectCountByEmail(String email);  
　　public User selectByEmailAndPass(User usr);  
  
}**

**\*子接口实现类（）**

**public class UserMapperImp implements UserMapper{  
　　　　public int selectCountByEmail(String email);  
　　public User selectByEmailAndPass(User usr);  
…..**

**…..  
}**

**2.** Comparator类

cmp.compare(arr[j], arr[j + 1]) > 0

3. Collections.sort

public static void main(String[] args) {  
  
ArrayList<Integer> arr=new ArrayList<>();  
arr.add(89);  
arr.add(78);  
arr.add(56);  
arr.add(92);  
arr.add(76);  
  
//升序排列；  
Collections.sort(arr);  
  
//逆序输出  
Collections.reverse(arr);  
for(int i=0;i<arr.size();i++){  
//这就是你要的效果  
System.out.println("0"+i+" "+arr.get(i));  
}  
  
}

对list值排序

# 3.logger.

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**private** **static** **final** Logger logger = LoggerFactory.getLogger(RobotSoftwareServiceImpl.**class**);

***错误格式参数logger***.error(format, arg);

***例子：***

***I=10，j=11***

***logger***.error(“I这个数是｛｝”, “j这个数是｛｝”, j);

当arg为异常对象时{}不会参数化，直接打印整个对象

# 4.获取当前项目路径

String path = this.getClass().getClassLoader().getResource("").getPath(); 这个是获取此文件编译后的路径地址

String root = System.*getProperty*(**"user.dir"**) 获取项目根目录

# 5字符串连接

String pathname = **new** StringBuilder().append(System.*getProperty*(**"user.dir"**)).append(**"/geektest-athena/src/main/resources/config/map.xml"**).toString();

# 6.路径分隔符

System.*getProperty*(**"file.separator"**))

Windows下\反斜杠，而且必须用[\\代表一个\](file:///\\代表一个\)表示路径

Linux下/斜杠表示路径

\*\*写路径时统一写反斜杠（兼容不同平台，相对路径即可）

# 7推薦使用class.getResource

操作的都是編譯后的文件即targeet下的class下的

URL url = Dom4jUtil.**class**.getResource(**"/config/map.xml"**);

pathname1=url.getfile();

File inputXml = **new** File(pathname1);

# 8数据库转java语言为空时的判断

必须是对象才能判断是否为空(大写，mapxml文件)

Double double1= sevice.getdoul()

If （!double1.equal(null)）{

double double1= double1. doubleValue()

}

9.变量初始化

Float a= 0f,

Double b= 0d,

10list 定义

List<String> robotLoc = **new** ArrayList<String>();

# 11转义字符

\*Split 用.分隔时需要转义；[\\.表示](file:///\\.表示).

\* \" 代表一个双引号字符”

例1

"\""+"172.16.42.93"+"\""打印结果就是“172.16.42.93”

Netty

*//心跳時間，如果0秒不讀，或者30秒不写就会调用userEventTriggered方法，重写改方法，让其写sendmessge即心跳*

//设置通道的各种配置pipeline()返回已经分配通道ChannelPipeline

**public class** APISocketClientChannelInitializer **extends** SocketClientChannelInitializer {  
 **static final int *BUFFER\_SIZE*** = 64 \* 1024;  
 **static final** String ***DELIMITER*** = **"\r\n"**;  
 @Override  
 **protected void** initChannel(Channel channel) **throws** Exception {  
 **socketClientHandler**= **new** APISocketClientHandler();  
 *//心跳時間，如果0秒不讀，或者30秒不写就会调用userEventTriggered方法，重写改方法，让其写sendmessge即心跳* channel.pipeline().addLast(**new** IdleStateHandler(0, 20, 0));

//设置读取，写入时的大小，以及分隔符bytebuf类型  
 channel.pipeline().addLast(**new** DelimiterBasedFrameDecoder(***BUFFER\_SIZE***, Unpooled.*copiedBuffer*(***DELIMITER***.getBytes(Charset.*forName*(**"UTF-8"**)))));  
 *//字符串解码器，读取得字符串转byteBuff传输* channel.pipeline().addLast(**new** StringDecoder());  
 *//字符串编码器，读取的时候byteBuff转string* channel.pipeline().addLast(**new** StringEncoder());  
// addLast()参数为**ChannelHandler一些处理io得事件或者拦截io的操作，并将其转发给通道的下一个处理程序。**

// **socketClientHandler包含sendmesge（string）需要编码成bytebuff，所以也要加入**  
 channel.pipeline().addLast(**socketClientHandler**);

JSONObject msg=(JSONObject) JSONObject.*toJSON*(robotTaskRequestMsg);

# 12类的序列化反序列化Serializable

**定义：**程序在运行时实例化出对象，这些对象存在于内存中，随着程序运行停止而消失，但如果我们想把某些对象（一般都是各不相同的属性）保存下来或者传输给其他进程（比如网络传输bean2json），在程序终止运行后这些对象仍然存在，可以在程序再次运行时读取这些对象的信息，或者在其他程序中利用这些保存下来的对象信息恢复成实例对象(json2bean)。这种情况下就要用到对象的序列化和反序列化

简而言之：就是暂时保存下来，以防丢失，string int类型都实现了该接口

\*Serializable接口没有方法和变量，只是一个可序列化标识，

\*实现该接口的类的子类也是可序列化

\*具有传递性，即实现该接口的类的成员变量也是可序列化的，即注意引用其他容器类时注意内存开销

\*实现该接口的类要定义一个long变量：serialVersionUID 用来表明类的不同版本间的兼容性:Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体（类）的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常

————————————————

private static final long serialVersionUID = 1L;

**解释：**序列化：将java对象保存到io流（二进制）或文件（xml，json对象）里，然后再远程传输（写）

反序列化：将io流文件还原成java对象使用，（读）

\*（json对象）序列化几种形式；json,xml

JSONObject msg=(JSONObject) JSONObject.*toJSON*(robotTaskRequestMsg);

*toJSON()方法就是将java对象序列化未json对象保存，（最终都是以二进制数据流传输）*

\*（二进制）可以用最原始的方法新建class实现Serializable接口(直接转二进制传输)

\*文件序列化利用ObjectInPutStream和ObjectOutPutSream。这两个类接收一个In/outPutStream参数

对象赋值，然后用soket ,获取客户段输入流getInputStream

Soket s=new soket(127.0.0.1,8888)

outputStream O=s.getoutputStream()

ObjectOutPutSream out= new ObjectOutPutSream(socket. getoutputStream()

)

Out.writeObject(student);写入

再用ObjectinPutSream in= new ObjectinPutSream(socket. getinputStream()

)

Student student=(student)In. readObject();反序列化

# 13json与javabean转化

com.alibaba.fastjson

//json2bean

方法2JSON.*toJavaObject*(jsonobject, CallbackHeader.**class**);

方法1.RobotTaskResponseBody data=JSON.*parseObject*(dataJson.toJSONString(),**new** TypeReference<RobotTaskResponseBody>(){});

//bean2jsonobject

.JSONObject msg=(JSONObject) JSONObject.*toJSON*(robotTaskRequestMsg);

//Jobject2string

JSONObject.toJSONString()

//jsonstring2list object

List<integer>lsit=JSONUtil.*parseObject*(JSONUtil.*toJSONString*(bodybean),List.class, Integer.class);

可用泛型获得指定类型的list,如下根据jsonstring获取指定类型的list

private <T extends Collection> T getCollection(String key, Class<T> collType, Class<?>[] elemTypes) {  
 return JSONUtil.*parseObject*(JSONUtil.*toJSONString*(this.get(key)), collType, elemTypes);  
}

//Jsonstring2bean

JSONUtil.*parseObject*(JSONUtil.*toJSONString*(this.map), TypeReference <student>);

可用泛型获取指定类型的bean

public <T> T getBody(TypeReference<T> type) {  
 return JSONUtil.*parseObject*(JSONUtil.*toJSONString*(this.map), type);  
  
}

# 14测试类的启动类（测试需要扫描用到的bean）

//貌似ComponentScan不管用，扫描不到bean，只能通过@SpringBootTest 的class属性添加

@ComponentScan(basePackages = **"com.geekplus.test.athena"**,excludeFilters={@ComponentScan.Filter(type= FilterType.***ASSIGNABLE\_TYPE***,value=MotanRpcConfiguration.**class**)})

//用几个加几个，并且如果用到的RobotTaskComponent.**class又用到别的bean也要加入**  
@SpringBootTest(classes = { MotanClient.**class**, RobotTaskComponent.**class**})

# 15泛型以及newinstanc初始化对象

//这是个泛型方法， 1输入参数必须是Class<T>类型即stuedent.class

2. T类必须继承AbstractMsg

3.返回值就是个T类型初始化对象如stuedent

<T **extends** AbstractMsg> T newInstance(Class<T> clz) { //输入参数为clasename.class

**try** {  
 **return** clz.newInstance();//  
 } **catch** (Exception e) {  
 ***log***.error(**"new instance for {} failed! cause:{}"**, clz, JSONUtil.*objToJson*(e));  
 }  
 **return null**;  
}

# 16json与实体对象转化时

@JsonInclude(value = Include.***NON\_NULL***)  
**public class** ResponseHeader **implements** Serializable {  
 **private static final long *serialVersionUID*** = 1L;  
  
 */\*\*  
 \* 对应{****@link*** *RequestHeader}中的#requestId字段  
 \*\*/* **private** String **responseId**;  
 */\*\*  
 \* 响应码  
 \*/* **private int code**;  
 */\*\*  
 \* 响应提示  
 \*/* **private** String **msg**;  
  
 */\*\*  
 \* i18n消息的参数列表（RMS内部字段）  
 \** ***@since*** *3.1.0  
 \*/* @JsonIgnore  
 **private** Object[] **args**;

**15判断e对象是否是**APIException类的实例化

如果是就强制转化ae

**if** (e **instanceof** APIException) {  
 APIException ae = (APIException) e;  
 respMsg.getHeader().setCode(ae.getCode());  
 respMsg.getHeader().setMsg(ae.getMsg());  
 respMsg.getHeader().setArgs(ae.getArgs());

# 15过滤器使用

**//查找**RMSServerConfig对象

**private** List<RMSServerConfig> **server**=**new** ArrayList<>();//定义对象list  
**public** RMSServerConfig getRMSServerConfig(String protocolType){  
 // Optional表示获取的是单个对象要想获取的是集合用List,

**//server**.stream()表示这个集合的所有对象，

// Optional<RMSServerConfig> serverConfigOptional=server.stream().filter(rmsServerConfig->protocolType.equals(rmsServerConfig.getProtocolType())).findfisrt()

然后获取RMSServerConfig对象

if(serverConfigOptional.isPresent()) {  
 RMSServerConfig rmsServerConfig = serverConfigOptional.get();  
 return rmsServerConfig;

方法表示filter过滤判断获得rmsServerConfig（RMSServerConfig实例化的）第一个对象,

如果获取多个对象为

List<A>firstA=AList.stream().filter(a->"hanmeimei".equals(a.getUserName())) .collect(Collectors.toList());

//单个对象获取用Optional<RMSServerConfig>.get()

Optional<RMSServerConfig>serverConfigOptional=**server**.stream().filter(rmsServerConfig->protocolType.equals(rmsServerConfig.getProtocolType())).findFirst();  
 **if**(serverConfigOptional.isPresent()) {//如果存在对象  
 RMSServerConfig rmsServerConfig = serverConfigOptional.get();（获取对象）  
 **return** rmsServerConfig;  
 }**else** {  
 **return null**;  
 }  
}

**MAP过滤器**

注：entrySet()返回一个包含所有key,value得集合，用于遍历，可以获取key和value

1. *//Map -> Stream -> Filter -> String*
2. String result = A\_MAP\_EXAMPLE.entrySet().stream()
3. .filter(map -> "something".equals(map.getValue()))
4. .map(map->map.getValue())
5. .collect(Collectors.joining());
6. *//Map -> Stream -> Filter -> MAP*
7. Map<Integer, String> collect = A\_MAP\_EXAMPLE.entrySet().stream()
8. .filter(map -> map.getKey() == 2)
9. .collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()));

**15.2循环遍历条件map（）方法**

抽取所有A对象名字叫lipeng的年龄集合

Fileter里的对象名随便器，例如e,但是map里的必须写返回的对象A

Map()跟fileter可以交叉使用，

List<A对象 > AList= new arrylist<>()

List<Long>idList=AList.stream().fileter(e->e.getname.equale(“lipeng”)).map(A::getage).collect(Collectors.toList());

----扩展\*\*循环操作myShapesCollection是个集合

在 JDK 8 以后，推荐使用聚合操作对一个集合进行操作。聚合操作通常和 lambda 表达式结合使用，让代码看起来更简洁（因此可能更难理解）。下面举几个简单的栗子：

1.使用流来遍历一个 ShapesCollection，然后输出红色的元素：

myShapesCollection.stream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

1

2

3

2.你还可以获取一个并行流（parallelStream），当集合元素很多时使用并发可以提高效率：

myShapesCollection.parallelStream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

1

2

3

3.聚合操作还有很多操作集合的方法，比如说你想把 Collection 中的元素都转成 String 对象，然后把它们 连起来：

注：map得作用是相当于遍历循环执行一个方法，是object（elements子元素）的方法，相当于object.tosttring(),然后调用collect方法，所有集合要做什么，如Collectors.joining(", ")拼接，Collectors.toList()转化为list集合这里肯定是list<string>

String joined = elements.stream()

.map(Object::toString) //

.collect(Collectors.joining(", "));

**15.3Collector和Collectors**

## 一、概述

Collector是专门用来作为Stream的collect方法的参数的。

public interface Stream<T> extends BaseStream<T, Stream<T>> {

<R, A> R collect(Collector<? super T, A, R> collector);

}

而Collectors是作为生产具体Collector的工具类。

## 二、Collector

Collector主要包含五个参数，它的行为也是由这五个参数来定义的，如下所示：

public interface Collector<T, A, R> {

// supplier参数用于生成结果容器，容器类型为A

Supplier<A> supplier();

// accumulator用于消费元素，也就是归纳元素，这里的T就是元素，它会将流中的元素一个一个与结果容器A发生操作

BiConsumer<A, T> accumulator();

// combiner用于两个两个合并并行执行的线程的执行结果，将其合并为一个最终结果A

BinaryOperator<A> combiner();

// finisher用于将之前整合完的结果R转换成为A

Function<A, R> finisher();

// characteristics表示当前Collector的特征值，这是个不可变Set

Set<Characteristics> characteristics();

}

Collector拥有两个of方法用于生成Collector实例，其中一个拥有上面所有五个参数，另一个四个参数，不包括finisher。

public interface Collector<T, A, R> {

// 四参方法，用于生成一个Collector，T代表流中的一个一个元素，R代表最终的结果

public static<T, R> Collector<T, R, R> of(Supplier<R> supplier,

BiConsumer<R, T> accumulator,

BinaryOperator<R> combiner,

Characteristics... characteristics) {/\*...\*/}

// 五参方法，用于生成一个Collector，T代表流中的一个一个元素，A代表中间结果，R代表最终结果，finisher用于将A转换为R

public static<T, A, R> Collector<T, A, R> of(Supplier<A> supplier,

BiConsumer<A, T> accumulator,

BinaryOperator<A> combiner,

Function<A, R> finisher,

Characteristics... characteristics) {/\*...\*/}

}

Characteristics：这个特征值是一个枚举，拥有三个值：CONCURRENT（多线程并行），UNORDERED（无序），IDENTITY\_FINISH（无需转换结果）。其中四参of方法中没有finisher参数，所以必有IDENTITY\_FINISH特征值。

## 三、Collectors

Collectors是一个工具类，是JDK预实现Collector的工具类，它内部提供了多种Collector，我们可以直接拿来使用，非常方便。

#### 5.6.1 toCollection

将流中的元素全部放置到一个集合中返回，这里使用Collection，泛指多种集合。

public class CollectorsTest {

public static void toCollectionTest(List<String> list) {

List<String> ll = list.stream().collect(Collectors.toCollection(LinkedList::new));

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

toCollectionTest(list);

}

}

#### 5.6.2 toList

将流中的元素放置到一个列表集合中去。这个列表默认为ArrayList。

public class CollectorsTest {

public static void toListTest(List<String> list) {

List<String> ll = list.stream().collect(Collectors.toList());

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

toListTest(list);

}

}

#### 5.6.3 toSet

将流中的元素放置到一个无序集set中去。默认为HashSet。

public class CollectorsTest {

public static void toSetTest(List<String> list) {

Set<String> ss = list.stream().collect(Collectors.toSet());

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

toSetTest(list);

}

}

#### 5.6.4 joining

joining的目的是将流中的元素全部以字符序列的方式连接到一起，可以指定连接符，甚至是结果的前后缀。

public class CollectorsTest {

public static void joiningTest(List<String> list){

// 无参方法

String s = list.stream().collect(Collectors.joining());

System.out.println(s);

// 指定连接符

String ss = list.stream().collect(Collectors.joining("-"));

System.out.println(ss);

// 指定连接符和前后缀

String sss = list.stream().collect(Collectors.joining("-","S","E"));

System.out.println(sss);

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

joiningTest(list);

}

}

执行结果：

1234567891101212121121asdaa3e3e3e2321eew

123-456-789-1101-212121121-asdaa-3e3e3e-2321eew

S123-456-789-1101-212121121-asdaa-3e3e3e-2321eewE

StringJoiner：这是一个字符串连接器，可以定义连接符和前后缀，正好适用于实现第三种joining方法。

#### 5.6.5 mapping

这个映射是首先对流中的每个元素进行映射，即类型转换，然后再将新元素以给定的Collector进行归纳。

public class CollectorsTest {

public static void mapingTest(List<String> list){

List<Integer> ll = list.stream().limit(5).collect(Collectors.mapping(Integer::valueOf,Collectors.toList()));

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

mapingTest(list);

}

}

实例中截取字符串列表的前5个元素，将其分别转换为Integer类型，然后放到一个List中返回。

#### 5.6.6 collectingAndThen

该方法是在归纳动作结束之后，对归纳的结果进行再处理。

public class CollectorsTest {

public static void collectingAndThenTest(List<String> list){

int length = list.stream().collect(Collectors.collectingAndThen(Collectors.toList(),e -> e.size()));

System.out.println(length);

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

collectingAndThenTest(list);

}

}

执行结果为：

8

#### 5.6.7 counting

该方法用于计数。

public class CollectorsTest {

public static void countingTest(List<String> list){

long size = list.stream().collect(Collectors.counting());

System.out.println(size);

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

countingTest(list);

}

}

结果：

8

#### 5.6.8 minBy/maxBy

生成一个用于获取最小/最大值的Optional结果的Collector。

public class CollectorsTest {

public static void maxByAndMinByTest(List<String> list){

System.out.println(list.stream().collect(Collectors.maxBy((a,b) -> a.length()-b.length())));

System.out.println(list.stream().collect(Collectors.minBy((a,b) -> a.length()-b.length())));

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

maxByAndMinByTest(list);

}

}

执行结果为：

Optional[212121121]

Optional[123]

#### 5.6.9 summingInt/summingLong/summingDouble

生成一个用于求元素和的Collector，首先通过给定的mapper将元素转换类型，然后再求和。

参数的作用就是将元素转换为指定的类型，最后结果与转换后类型一致。

public class CollectorsTest {

public static void summingTest(List<String> list){

int i = list.stream().limit(3).collect(Collectors.summingInt(Integer::valueOf));

long l = list.stream().limit(3).collect(Collectors.summingLong(Long::valueOf));

double d = list.stream().limit(3).collect(Collectors.summingDouble(Double::valueOf));

System.out.println(i +"\n" +l + "\n" + d);

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

summingTest(list);

}

}

执行结果为：

1368

1368

1368.0

#### 5.6.10 averagingInt/averagingLong/averagingDouble

生成一个用于求元素平均值的Collector，首选通过参数将元素转换为指定的类型。

参数的作用就是将元素转换为指定的类型，求平均值涉及到除法操作，结果一律为Double类型。

public class CollectorsTest {

public static void averagingTest(List<String> list){

double i = list.stream().limit(3).collect(Collectors.averagingInt(Integer::valueOf));

double l = list.stream().limit(3).collect(Collectors.averagingLong(Long::valueOf));

double d = list.stream().limit(3).collect(Collectors.averagingDouble(Double::valueOf));

System.out.println(i +"\n" +l + "\n" + d);

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

averagingTest(list);

}

}

执行结果为：

456.0

456.0

456.0

#### 5.6.11 reducing

reducing方法有三个重载方法，其实是和Stream里的三个reduce方法对应的，二者是可以替换使用的，作用完全一致，也是对流中的元素做统计归纳作用。

public final class Collectors {

// 无初始值的情况，返回一个可以生成Optional结果的Collector

public static <T> Collector<T, ?, Optional<T>> reducing(BinaryOperator<T> op) {/\*...\*/}

// 有初始值的情况，返回一个可以直接产生结果的Collector

public static <T> Collector<T, ?, T> reducing(T identity, BinaryOperator<T> op) {/\*...\*/}

// 有初始值，还有针对元素的处理方案mapper，生成一个可以直接产生结果的Collector，元素在执行结果操作op之前需要先执行mapper进行元素转换操作

public static <T, U> Collector<T, ?, U> reducing(U identity,

Function<? super T, ? extends U> mapper,

BinaryOperator<U> op) {/\*...\*/}

}

实例：

public class CollectorsTest {

public static void reducingTest(List<String> list){

System.out.println(list.stream().limit(4).map(String::length).collect(Collectors.reducing(Integer::sum)));

System.out.println(list.stream().limit(3).map(String::length).collect(Collectors.reducing(0, Integer::sum)));

System.out.println(list.stream().limit(4).collect(Collectors.reducing(0,String::length,Integer::sum)));

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

reducingTest(list);

}

}

Optional[13]

9

13

效果可参见[Java流式操作系列-Stream](Java%E6%B5%81%E5%BC%8F%E6%93%8D%E4%BD%9C%E7%B3%BB%E5%88%97-Stream.md#4.4-reduce)

#### 5.6.12 groupingBy

这个方法是用于生成一个拥有分组功能的Collector，它也有三个重载方法：

public final class Collectors {

// 只需一个分组参数classifier，内部自动将结果保存到一个map中，每个map的键为?类型（即classifier的结果类型），值为一个list，这个list中保存在属于这个组的元素。

public static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(

Function<? super T, ? extends K> classifier) {/\*...\*/}

// 在上面方法的基础上增加了对流中元素的处理方式的Collector，比如上面的默认的处理方法就是Collectors.toList()

public static <T, K, A, D>Collector<T, ?, Map<K, D>> groupingBy(

Function<? super T, ? extends K> classifier,Collector<? super T, A, D> downstream) {/\*...\*/}

// 在第二个方法的基础上再添加了结果Map的生成方法。

public static <T, K, D, A, M extends Map<K, D>>

Collector<T, ?, M> groupingBy(Function<? super T, ? extends K> classifier,

Supplier<M> mapFactory,

Collector<? super T, A, D> downstream) {/\*...\*/}

}

实例：

public class CollectorsTest {

public static void groupingByTest(List<String> list){

Map<Integer,List<String>> s = list.stream().collect(Collectors.groupingBy(String::length));

Map<Integer,List<String>> ss = list.stream().collect(Collectors.groupingBy(String::length, Collectors.toList()));

Map<Integer,Set<String>> sss = list.stream().collect(Collectors.groupingBy(String::length,HashMap::new,Collectors.toSet()));

System.out.println(s.toString() + "\n" + ss.toString() + "\n" + sss.toString());

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

groupingByTest(list);

}

}

执行结果为：

{3=[123, 456, 789], 4=[1101], 5=[asdaa], 6=[3e3e3e], 7=[2321eew], 9=[212121121]}

{3=[123, 456, 789], 4=[1101], 5=[asdaa], 6=[3e3e3e], 7=[2321eew], 9=[212121121]}

{3=[123, 456, 789], 4=[1101], 5=[asdaa], 6=[3e3e3e], 7=[2321eew], 9=[212121121]}

groupingBy方法还有并发版的groupingByConcurrent，功能基本一致，只是返回的Collector是并行的。

#### 5.6.13 partitioningBy

该方法将流中的元素按照给定的校验规则的结果分为两个部分，放到一个map中返回，map的键是Boolean类型，值为元素的列表List。

该方法有两个重载方法：

public final class Collectors {

// 只需一个校验参数predicate

public static <T>

Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> predicate) {/\*...\*/}

// 在上面方法的基础上增加了对流中元素的处理方式的Collector，比如上面的默认的处理方法就是Collectors.toList()

public static <T, D, A>

Collector<T, ?, Map<Boolean, D>> partitioningBy(Predicate<? super T> predicate,

Collector<? super T, A, D> downstream) {/\*...\*/}

}

实例：

public class CollectorsTest {

public static void partitioningByTest(List<String> list){

Map<Boolean,List<String>> map = list.stream().collect(Collectors.partitioningBy(e -> e.length()>5));

Map<Boolean,Set<String>> map2 = list.stream().collect(Collectors.partitioningBy(e -> e.length()>6,Collectors.toSet()));

System.out.println(map.toString() + "\n" + map2.toString());

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

partitioningByTest(list);

}

}

执行结果：

{false=[123, 456, 789, 1101, asdaa], true=[212121121, 3e3e3e, 2321eew]}

{false=[123, 456, 1101, 789, 3e3e3e, asdaa], true=[212121121, 2321eew]}

#### 5.6.14 toMap

toMap方法是根据给定的键生成器和值生成器生成的键和值保存到一个map中返回，键和值的生成都依赖于元素，可以指定出现重复键时的处理方案和保存结果的map。

public final class Collectors {

// 指定键和值的生成方式keyMapper和valueMapper

public static <T, K, U>

Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,

Function<? super T, ? extends U> valueMapper) {/\*...\*/}

// 在上面方法的基础上增加了对键发生重复时处理方式的mergeFunction，比如上面的默认的处理方法就是抛出异常

public static <T, K, U>

Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,

Function<? super T, ? extends U> valueMapper,

BinaryOperator<U> mergeFunction) {/\*...\*/}

// 在第二个方法的基础上再添加了结果Map的生成方法。

public static <T, K, U, M extends Map<K, U>>

Collector<T, ?, M> toMap(Function<? super T, ? extends K> keyMapper,

Function<? super T, ? extends U> valueMapper,

BinaryOperator<U> mergeFunction,

Supplier<M> mapSupplier) {/\*...\*/}

}

实例：

public class CollectorsTest {

public static void toMapTest(List<String> list){

Map<String,String> map = list.stream().limit(3).collect(Collectors.toMap(e -> e.substring(0,1),e -> e));

Map<String,String> map1 = list.stream().collect(Collectors.toMap(e -> e.substring(0,1),e->e,(a,b)-> b));

Map<String,String> map2 = list.stream().collect(Collectors.toMap(e -> e.substring(0,1),e->e,(a,b)-> b,HashMap::new));

System.out.println(map.toString() + "\n" + map1.toString() + "\n" + map2.toString());

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

toMapTest(list);

}

}

执行结果：

{1=123, 4=456, 7=789}

{a=asdaa, 1=1101, 2=2321eew, 3=3e3e3e, 4=456, 7=789}

{a=asdaa, 1=1101, 2=2321eew, 3=3e3e3e, 4=456, 7=789}

第一种方式中，如果不添加limit限制，就会抛出异常。

还有并发的版本：toConcurrentMap,同样三种重载方法，与toMap基本一致，只是它最后使用的map是并发Map:ConcurrentHashMap。

#### 5.6.15 summarizingInt/summarizingLong/summarizingDouble

这三个方法适用于汇总的，返回值分别是IntSummaryStatistics，LongSummaryStatistics，DoubleSummaryStatistics。

在这些返回值中包含有流中元素的指定结果的数量、和、最大值、最小值、平均值。所有仅仅针对数值结果。

public class CollectorsTest {

public static void summarizingTest(List<String> list){

IntSummaryStatistics intSummary = list.stream().collect(Collectors.summarizingInt(String::length));

LongSummaryStatistics longSummary = list.stream().limit(4).collect(Collectors.summarizingLong(Long::valueOf));

DoubleSummaryStatistics doubleSummary = list.stream().limit(3).collect(Collectors.summarizingDouble(Double::valueOf));

System.out.println(intSummary.toString() + "\n" + longSummary.toString() + "\n" + doubleSummary.toString());

}

public static void main(String[] args) {

List<String> list = Arrays.asList("123","456","789","1101","212121121","asdaa","3e3e3e","2321eew");

summarizingTest(list);

}

}

执行结果：

IntSummaryStatistics{count=8, sum=40, min=3, average=5.000000, max=9}

LongSummaryStatistics{count=4, sum=2469, min=123, average=617.250000, max=1101}

DoubleSummaryStatistics{count=3, sum=1368.000000, min=123.000000, average=456.000000, max=789.000000}

最后我们可以从返回的汇总实例中获取到想要的汇总结果。

## 四、总结

整个Collectors工具类就是在为Collector服务，用于创建各种不同的Collector。部分功能与Stream中的方法重合了，为了简化代码，完全不必采用Collectors实现，优先Stream方法。

作者：唯一浩哥  
链接：https://www.jianshu.com/p/7eaa0969b424  
来源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

---------------------

# 16@ PostConstruct

* **使用注解： @PostConstruct 注解一个非静态的void方法**
* **效果：在Bean初始化之后（构造方法和@Autowired之后）执行指定操作。经常用在将构造方法中的动作延迟。**
* **备注：Bean初始化时候的执行顺序： 构造方法 -> @Autowired -> PostConstruct注解的方法**
* **如果想在生成对象时完成某些初始化操作，而偏偏这些初始化操作又依赖于依赖注入，那么就无法在构造函数（构造方法 -> @Autowired,即初始化时还没注入aut对象）中实现。为此，可以使用@PostConstruct注解一个方法来完成初始化，**

举个栗子：

public Class AAA {

    @Autowired

    private BBB b;

    public AAA() {

        System.out.println("此时b还未被注入: b = " + b);

    }

    @PostConstruct

    private void init() {

        System.out.println("@PostConstruct将在依赖注入完成后被自动调用: b = " + b);

    }

}

---------------------

# 17timer起线程轮询事件

**import** java.util.Timer;//定时器  
**import** java.util.TimerTask;

**private static** Timer *timer* = **new** Timer(**"api-callback"**)//初始化timer对象命名为api—

* *\*timer.* Schedule( java.util.TimerTask task, long delay,long period),timertaks继承runnale,起线程重写@run方法写自己要实现的功能，delay：当该bean在加载后延迟时间才轮询，跟**@PostConstruct一起用，**period：轮询的间隔时间

private void init() {  
 // 系统启动10秒后，每200毫秒调用一次callback方法  
 timer.schedule(new TimerTask() {  
 @Override  
 public void run() {  
 callback();  
 }  
 }, 10\_000, 200);  
}

# 17sutdent.class.getname,getsinglename区别

\* **getname=com.geekplus.student**

**\*getsinglename= student**

# 18匿名内部类（该类只初始化一次，而且没有名字）

匿名内部类的创建格式如下：可以是父类（），也可以是接口，类体实现接口的方法（该类只调用一次）

new 父类构造器（参数列表）|实现接口（）

{

//匿名内部类的类体部分

}

在上面的代码中，connect方法中在实例化ZooKeeper对象时用到了匿名内部类：

zoo = new ZooKeeper(host,5000,new Watcher() {

public void process(WatchedEvent we) {

if (we.getState() == KeeperState.SyncConnected) {

connectedSignal.countDown();

}

}

});

这个内部类没有自己的名字，而是用到了Watcher接口，而通常情况下接口是不能用new的，但是在匿名内部类中可以这样。匿名内部类的类体是一个名为process的方法，这个方法就是用来实现Watcher接口中定义的process抽象方法的。

# 19枚举类（

**一般作为数据库状态码与字段的映射；不能new初始化，只能调用方法获取）**

**\*每个枚举成员看成是枚举类的实例，所以定义有成员变量的枚举成员时，必须定义枚举类的有参构造函数的（顺序一定要对应上），枚举成员都是默默认public,static, final修饰**

**\*获取枚举的属性成员变量时用 枚举.getproperty(),**

**多属性枚举：**

**public enum APIResult {**

***OK*(0, APII18nConstant.*API\_RESULT\_OK*),  
*OK\_PART*(0, APII18nConstant.*API\_RESULT\_OK\_PART*)；**

**private int code;  
private String msg;  
  
APIResult(int code, String msg) {  
 this.code = code;  
 this.msg = msg;  
}  
  
public int getCode(  
) {  
 return code;  
}  
  
public String getMsg(  
) {  
 return msg;  
}**

**例1:**

**\*枚举最终获取的是**DirectionType 为***EAST***

***\*枚举可以有属性（int）,此属性必须被定义成一个变量，例如*direction，且必须定义一个以direction作为参数传入构造函数（即每个枚举如:** D**irectionType**.***EAST***(0)**就是一个初始化的对象，是通过构造函数得到的）;可以定义一个方法获取枚举类型，通过入参direction来获取，**DirectionType directiontype = DirectionType .valueOfDir(**3**)

**\*** valueOfDir （）方法根据输入的数字（或者不加枚举属性，根据字段名字）返回对应的枚举（判断客户端传来的数据是否为枚举中的一个，作出判断），*values获取所有枚举,* ent.**direction相当于**DirectionType.***EAST.getD*irection=0**

**package** com.geekplus.test.athena.controller;  
  
  
  
*/\*\*  
 \* <p>  
 \* 上北下南，左西右东  
 \* </p>  
 \*  
 \** ***@author*** *liujun 2019/2/26  
 \** ***@since*** *3.3.0  
 \*/***public enum** DirectionType {  
 */\*\*  
 \* 东  
 \*/* ***EAST***(0),  
 */\*\*  
 \* 南  
 \*/* ***SOUTH***(1),  
 */\*\*  
 \* 西  
 \*/* ***WEST***(2),  
 */\*\*  
 \* 北  
 \*/* ***NORTH***(3),  
 */\*\*  
 \* 无效  
 \*/* ***NONE***(-1);  
  
 */\*\*  
 \* 0:e 1:s 2:w 3:n  
 \*/* **private final int direction**;  
  
 DirectionType(**int** direction) {  
 **this**.**direction** = direction;  
 }  
  
 **public int** getDirection() {  
 **return direction**;  
 }  
  
 */\*\*  
  
 \* 获取方向所对应的枚举  
 \*  
 \** ***@param dir*** *方向（0:e 1:s 2:w 3:n）  
 \** ***@return*** *DirectionType  
 \** ***@author*** *liujun Oct 24, 2017  
 \*/* **public static** DirectionType valueOfDir(**int** dir) {  
 **for** (DirectionType ent : *values*()) {  
 **if** (ent.**direction** == dir) {  
 **return** ent;  
 }  
 }  
 **return *NONE***;  
 }  
  
 **public static void** main(String[] args) {  
 DirectionType directionType= *valueOfDir*(1);

//输出***SOUTH***  
 System.***out***.println(directionType);  
 }  
}

# 20域对象

**(包含一系列有序，有逻辑的对象的操作统称,生命周期由可能几个对象几个操作完成)**

例1：HttpSession 域

1.生命周期：

    在第一次调用request.getSession()方法时，服务器会检查是否已经有对应的session，如果没有就在内存中创建一个session并返回。

（1）当一段时间内session没有被使用（默认为30分钟），则服务器会销毁该session。

（2）如果服务器非正常关闭，没有到期的session也会跟着销毁。

（3）如果调用session提供的invalidate()，可以立即销毁session。

用户打开浏览器访问，创建session（开始），session超时或者被声明失效，该对象生命周期结束；

2.作用范围：一次会话。

HttpSession 在服务器中，为浏览器创建独一无二的内存空间，在其中保存会话相关的信息

注意：服务器正常关闭，再启动，Session对象会进行钝化和活化操作。同时如果服务器钝化的时间在session 默认销毁时间之内， 则活化后session还是存在的。否则Session不存在。  如果JavaBean 数据在session钝化时，没有实现Serializable 则当Session活化时，会消失。

# 21套接字

**（12y7.0.0.1：8080）**

[编辑](javascript:;) [讨论](https://baike.baidu.com/planet/talk?lemmaId=9637606)

TCP用主机的IP地址加上主机上的端口号作为TCP连接的端点，这种端点就叫做套接字（socket）或插口。

套接字用（IP地址：端口号）表示。

它是网络通信过程中端点的抽象表示，包含进行网络通信必需的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

# 22 ByteBuf读取io.netty.buffer;

**(一系列有序的0个或多个byte字节流,网络传输都是java数据转换成字节流，然后再解析buf.tostring（）)**

\*\*ByteBuf buf = (ByteBuf) msg; 获取字节流

**byte**[] headerBytes = **new byte**[2];定义二进制数组  
 buf.readBytes(headerBytes, 0, 2);从buf流第一个位置读取2连个字节，读两个字节到headerBytes从headerBytes下表为0的位置，假如buf为10个长度，那么读完就剩8个了，

*//读数据长度，从剩余8个里读* **byte**[] dataLenBytes = **new byte**[2];  
 buf.readBytes(dataLenBytes, 0, 2);

readBytes（目的数组，读到目的数组的初始位置，读两个字节）

**private class** ChannelHandler **extends** ChannelInboundHandlerAdapter {  
 @Override  
 **public void** channelRead(ChannelHandlerContext ctx, Object msg) {  
 ByteBuf buf = (ByteBuf) msg;  
 InetSocketAddress addr = getAddressOfCtx(ctx);  
  
 **try** {  
 **int** bufLength = buf.readableBytes();  
 **if** (bufLength < 8) {  
 ***ROBOT\_SERVER\_LOGGER***.error(**"channelRead-error dataBody error bufLength<4 bufLength:{} ip:{} port:{}"**,  
 bufLength, addr.getHostName(), addr.getPort());  
 **return**;  
 }  
  
 *//读帧头* **byte**[] headerBytes = **new byte**[2];  
 buf.readBytes(headerBytes, 0, 2);  
 **char** header = BitConverter.*toChar*(headerBytes, 0);  
 *//读数据长度* **byte**[] dataLenBytes = **new byte**[2];  
 buf.readBytes(dataLenBytes, 0, 2);  
 *// 读机器人编号* **byte**[] robotIdBytes = **new byte**[4];  
 buf.readBytes(robotIdBytes, 0, 4);  
 **int** robotId = BitConverter.*toInt*(robotIdBytes, 0);  
 *// 8为机器人编号的长度加上帧头及数据长度，向业务中返回的byte数组* **byte**[] req = **new byte**[bufLength - 8];  
 *// 读指令内容* buf.readBytes(req, 0, req.**length**);

# 23二进制

1．Byte 字节，是计算机文件大小的基本计算单位；1byte=8bit ，

例子：01001111 一个字节可以代表256个内容，所有字母加符号都没有256

个。

例2：一个汉字占2字节 字母和数字占一个字节

2.Bit 比特，二进制，0或1 ，二进制最小单位

一  char和string的区别：

     1  char是表示的是字符，定义的时候用单引号，只能存储一个字符。例如; char='d'.

         而String表示的是字符串，定义的时候用双引号，可以存储一个或者多个字符。例如：String=“we  are neuer”。

    2   char是基本数据类型，而String是个类，属于引用数据类型。String类可以调用方法，具有面向对象的特征。

二   char类型

     char在Java中是16位的，因为Java用的是Unicode。

例1

**package** com.geekplus.test.athena.task;  
  
**import** com.alibaba.druid.sql.visitor.functions.Char;  
  
**import** java.io.UnsupportedEncodingException;  
  
**public class** bte {  
 **public char c**=**'A'**;  
*//默认返回ascim* **public byte**[] **b**;  
*// 定义 char byte时，用单引号* **public byte abyte**=**'1'**; *//byte类型值就是1对应的ascii码49，只能定义一位，不能定义汉字，只能定义英文编码  
// public byte abyte='21'; //只能定义一位，会报错，一个字节* **public byte**[] **bb**=**new byte**[2];  
  
 **public char aChar**=**'丁'**;*//char为unicode编码，占用两个字节，汉字占用两个字节;只能定义一个汉字，两个字母，两个数字* {  
 **try** {  
 **b** = **"AB"**.getBytes(**"utf-8"**);  
 } **catch** (UnsupportedEncodingException e) {  
 e.printStackTrace();  
 }  
 }  
  
  
 *// char转换byte* **public byte**[] charToByte() {  
  
 **byte**[] byte1 = **new byte**[2];  
 byte1[0] = (**byte**) ((**c** & 0xFF00) >> 8);  
 byte1[1] = (**byte**) (**c** & 0xFF);  
  
 **for** (**byte** bt :byte1){  
 System.***out***.println(**"charToByte:"**+bt);  
 *//s输出A的asscii码 0,65,* }  
 **return** byte1;  
  
 }  
 *//字节转换string* **public** String Byte2String() {  
 *//将byte[]数组转换为 string* String str= **new** String (**b**);  
  
 System.***out***.println(**" Byte2String:"**+str);  
 *//AB的字节数组为byte[] bte={65,66},输出string为AB* **return** str;  
  
 }  
  
 *//• byte转换为char：* **public char** byteToChar() {  
 **char** char1 = (**char**) (((**b**[0] & 0xFF) << 8) | (**b**[1] & 0xFF));  
 System.***out***.println(**"byteToChar:"**+char1);  
 **return** char1;  
  
  
 }  
 *//• byte转换为char：* **public void** initBytes() {  
 **for** (**int** i=0;i<**bb**.**length**;i++){  
 *//初始化string为byte类型* **bb**[i]=**new** Byte(**"65"**);  
 System.***out***.println(**"initBytes"**+i+**":"**+**bb**[i]);  
 *//输出65* }  
 }  
  
 **public static void** main(String[] args) {  
 bte bt= **new** bte();  
 bt.charToByte();  
 bt.byteToChar();  
 bt.Byte2String();  
 bt.initBytes();  
 System.***out***.println(**"aChar 的char值："**+bt.**aChar**+**";abyte 的byte值："**+bt.**abyte**);  
  
 }

输出结果：

charToByte:0

charToByte:65

byteToChar:䅂

Byte2String:AB

initBytes0:65

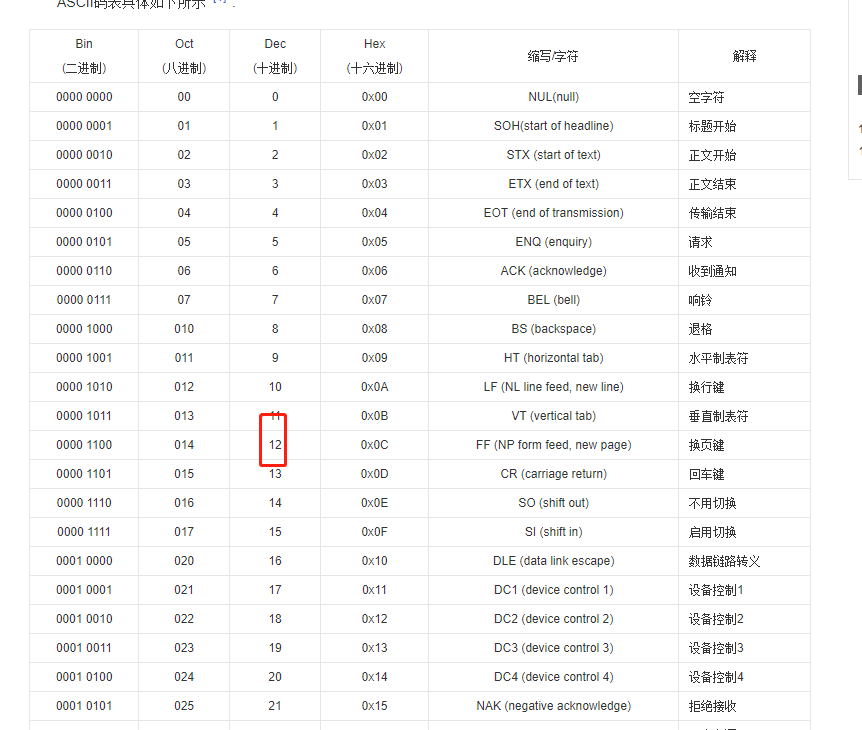
initBytes1:65

aChar 的char值：丁;abyte 的byte值：49  
}

# 24.编码格式

**1ASCII码**范围（十进制）0-128，一个码占用一个字节(byte)代表英文的编码，英文，数字，特殊符号

例子：A对应的65，getbyte也是65（ASCII）

**2Unicode（兼容ascii）两个字节最高位都为0**

[编辑](javascript:;) [讨论](https://baike.baidu.com/planet/talk?lemmaId=750500)

Unicode（[统一码](https://baike.baidu.com/item/%E7%BB%9F%E4%B8%80%E7%A0%81)、万国码、单一码）是计算机科学领域里的一项业界标准，包括字符集、编码方案等。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的[二进制](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6)编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990年开始研发，1994年正式公布。[1]

Unicode通常用两个字节表示一个字符，可以表示6万多种字符，原有的英文编码从单字节变成双字节，只需要把高字节全部填为0就可以, 在表示一个Unicode的字符时，通常会用“U+”然后紧接着一组十六进制的数字来表示这一个字符,

中为范围4E00-9FA5



丁的表示方法为4e01,十进制为19969

# 26.arraylist(map )

List<mapcell> list=null，是错误的，必须初始化，否则空指针

正确：List<mapcell>=new arraylist<mapcell>()

# 27.pagehelper分业功能

**应该在services层实现，要不然分业不成功**

*pageNum 当前展示的页码，  
 \* 当用pagehelper分页时必须放在service层实现，要不然无法分页，返回对象PageInfo<>  
\* \*/* @Override  
 public PageInfo<BaseShelf> getShlefListCondtion(int pageNum,String shelfCode,int startPlacementIndexX ,int startPlacementIndexY ,int endPlacementIndexX ,int endPlacementIndexY) {  
 List<BaseShelf> shelfList;  
 PageHelper.*startPage*(pageNum,10);  
 BaseShelfExample example = new BaseShelfExample();  
 if (shelfCode.equals("none")) {  
  
 example.or().andPlacementXLessThanOrEqualTo(endPlacementIndexX).andPlacementXGreaterThan(startPlacementIndexX).andPlacementYGreaterThan(startPlacementIndexY).andPlacementYLessThan(endPlacementIndexY);  
  
 } else {  
 example.or().andPlacementXLessThanOrEqualTo(endPlacementIndexX).andPlacementXGreaterThan(startPlacementIndexX).andPlacementYGreaterThan(startPlacementIndexY).andPlacementYLessThan(endPlacementIndexY).andShelfCodeLike(shelfCode);  
 }  
 shelfList=baseShelfMapper.selectByExample(example);  
 PageInfo<BaseShelf> pageInfo= new PageInfo<>(shelfList);  
  
 if (!shelfList.isEmpty()) {  
 return pageInfo;  
  
  
 }  
 return null;

# 28.配置文件的属性的读取赋值给bean

@Configuration

public class NettyServerConfiguration {

private static final String MONITOR\_SERVER\_PREFIX = "netty.server.monitor-server";

public static final String HTTP\_API\_PREFIX = "netty.server.http-api";

public static final String SOCKET\_API\_PREFIX = "netty.server.socket-api";

public static final String WEBSOCKET\_API\_PREFIX = "netty.server.websocket-api";

public static final String ROBOT\_SERVER\_PREFIX = "netty.server.robot";

public static final String CHARGER\_SERVER\_PREFIX = "netty.server.charger-server";

private static final String STOPBUTTON\_SERVER\_PREFIX = "netty.server.stop-button";

/\*\*

\* 构造monitor-server

\*

\* @return

\* @author liujun 2018年12月24日

\* @since 3.2.0

\*/

@Bean(initMethod = "init", destroyMethod = "stop")

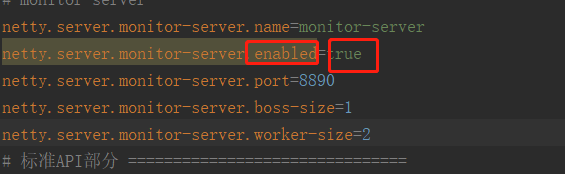
// ConditionalOnProperty根据配置文件对象属性值判断是否一致决定monitorController配置bean是否生效，是否注入这个bean）

这里变量MONITOR\_SERVER\_PREFIX =netty.server.monitor-serverprefix

如下的意思为：

配置文件中的netty.server.monitor-serverprefix（对应prefix）对象的enabled属性（对应value）的值是否等于true（对应havingvalue）;如果配置文件是true那么就注册这个bean,下图中确实为true，：

知识点：不想启用时，改成false，灵活注册bean



@ConditionalOnProperty(prefix = MONITOR\_SERVER\_PREFIX, value = "enabled", havingValue = "true")

// ConfigurationProperties：用来注解类或配置类的bean,即。Properties或yaml配置文件的一组属性，bean在。Properties文件中查找MONITOR\_SERVER\_PREFIX(netty.server.monitor-server:注意此类一定要有setter,getter方法，否则属性绑定失败)类对应的属性，如下：



初始化给NettyServer(monitorServerHandler)对象，支持配置文件属性为a-b对应实体属性aB.

\*如果绑定的是静态属性，setter方法必须是非静态方法,因为ConfigurationProperties只能调用类的非静态setter方法

\*当你用这个bean时就已经有了这些属性

@ConfigurationProperties(MONITOR\_SERVER\_PREFIX)

public NettyServer monitorController(MonitorServerHandler monitorServerHandler) {

return new NettyServer(monitorServerHandler);

}

@Bean

@ConditionalOnProperty(prefix = MONITOR\_SERVER\_PREFIX, value = "enabled", havingValue = "true")

public MonitorServerHandler monitorServerChannelInitializer() {

return new MonitorServerHandler();

}

--------------接着上边的配置类讲，配置类把bean实现了注入，---如上第一个配置bean，名字叫monitorController（方法名）的NettyServer（方法返回值）类型的bean，接下来的接口就是怎么获取所有注入的bean----------------------------------------------------------

@Component  
public class NettyServerManager implements ApplicationContextAware {  
 private Map<String, NettyServer> servers = new HashMap<>();

@Override  
public void setApplicationContext(ApplicationContext applicationContext) {  
 Map<String, NettyServer> beans = applicationContext.getBeansOfType(NettyServer.class);  
 for (NettyServer val : beans.values()) {  
 this.servers.put(val.getName(), val);  
 }  
}

实现了ApplicationContextAware接口，重写setApplicationContext方法，获取指定对象类型的bean如(NettyServer.class)类型的，

applicationContext.getBeansOfType(NettyServer.class)方法获取所有注入得NettyServer类型的bean;疑问为什么不直接赋值：再重新取出来赋值

# 28. @SafeVarargs可变长度注解

引用@**SafeVarargs注解后,改方法可以填写可变长度个int类型的参数**

如下可以求和sum(1,2,3)

1. @safeVarargs
2. public int sum(int... args) {
3. int result = 0;
4. for (int value : args) {
5. result += value;
6. }
7. return result;
8. }
9. 也可以返回泛型数组
10. @SafeVarargs  
    private static <T> T[] array(T... values) {  
     return values;  
    }

# 28. ApplicationEventPublisherAware事件发布

**此接口可以实现解耦，即在操作成功后发布事件，然后监听者收到后执行相关方法，**

# 29.websokect初始化器

@Override  
 protected void initChannel(SocketChannel ch) {  
 ChannelPipeline pipeline = ch.pipeline();  
 //ByteBuf->HttpRequest或HttpContent和HttpResponse或者HttpContent->ByteBuf，即HTTP请求的解码和编码  
 pipeline.addLast(new HttpServerCodec());  
 //把多个消息转换为一个单一的FullHttpRequest或是FullHttpResponse，原因是HTTP解码器会在每个HTTP消息中生成多个消息对象HttpRequest/HttpResponse,HttpContent,LastHttpContent  
 pipeline.addLast(new HttpObjectAggregator(*BUFFER\_SIZE*));  
//大文件支持，即允许发送大文件  
 pipeline.addLast(new ChunkedWriteHandler());  
 //WebSocketServerCompressionHandler：WebSocket数据压缩(可选)  
 pipeline.addLast(new WebSocketServerCompressionHandler());  
  
 pipeline.addLast(new WebSocketServerProtocolHandler("/", null, true, *BUFFER\_SIZE*));  
 pipeline.addLast(new MonitorChannelHandler());  
 *log*.info("MonitorServer - a connection is ready. SocketChannel:{}", ch);  
 }

MonitorChannelHandler业务处理handler，监听，写方法,继承SimpleChannelInboundHandler<TextWebSocketFrame>泛型，指定类型后，即为channelRead0接收类型

private class MonitorChannelHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame frame) {  
 // 消息转换  
 JsonNode jsonNode = JSONUtil.*jsonToNode*(frame.text());  
 Class<? extends AbstractRequestMsg> clz =  
 monitorMsgFactory.getRequestMsgClass(jsonNode.get("msgType").asText());  
 if (clz == null) {  
 *log*.warn("unknown monitor request msg. msg:{}", frame.text());  
 sendMsg(ctx, monitorMsgFactory.newDefaultResponseMsg(null, APIResult.*ERROR*));  
 return;  
 }  
 AbstractRequestMsg reqMsg = JSONUtil.*nodeToObj*(jsonNode, clz);  
  
 // 消息校验  
 validMsg(ctx, reqMsg);  
 // 消息发送  
 sendMsg(ctx, handlerContext.execute(ctx, reqMsg));  
 }

private void sendMsg(ChannelHandlerContext ctx, AbstractResponseMsg respMsg) {  
 if (ctx == null || respMsg == null) {  
 return;  
 }  
  
 ctx.writeAndFlush(new TextWebSocketFrame(JSONUtil.*objToJson*(respMsg)));  
}

# 30.静态变量static（可以用来做资源缓存）

**1可通过类名.静态方法.静态变量访问**

**2静态方法只能使用静态变量**

**3静态变量不是必须初始化，有默认值，0或null（类加载后，先初始化静态变量/静态块，且仅初始化一次->）**

**4所谓静态是指他是类变量，不随实例方法的调用，退出而改变销毁这个变量的值，可以改变他的值，跟final区别，final修饰的是常量，最终的，不允许改变值**

**两种动态初始化静态变量**

**a静态块中初始化操作，给静态变量赋值，一次性的。**

**b命名静态变量为一个泛型对象如，list,hashmap,然后可以动态往集合增加删除元素，(做资源的缓存,即程序运行期间，把数据永久保存下来)**

*Private static Logger logger= LoggerFactory.getLogger(WebSocketClient.class);  
 public static Map<String,String> jsonMap=new HashMap<>();  
// public map  
 @OnOpen  
 public void onOpen(Session session) {  
 logger.info("WebSocketClient connect success");  
 }  
  
 @OnMessage  
 public void onMessage(String message) {  
 logger.info("receive success: {}",message);  
 this.addInfo(message);  
  
 }  
  
 @OnClose  
 public void onClose() {  
 logger.info("connection closed");  
  
 }  
 /\* @OnError  
 public void onError() {  
  
 logger.info("something erro occur");  
 }\*/  
  
  
 public static void addInfo(String message){  
 jsonMap.put("jsonMap",message);  
 /\* Map<String, String> map = new HashMap<>();  
  
 this.setJsonMap(new HashMap<>(){"jsonMap"，message});\*/  
 }*

# 31.抽象类实现接口思想（）

当设计一个类只需要实现一个接口的的个别方法时，先用抽象类实现接口（不用写实现方法），然后用普通类继承该抽象类，并实现自己想实现的接口方法即可

# 32. equalsignorecase和equals区别（）

Equalsignorecase忽略大小写，equals区分大小写

# 33. 回调类的实现（APICallbackRouter）

**（事件发生时告诉上游已经完成）**

0，定义可排序队列，可用来保存要发送的消息

1@postconstrct定时任务，当系统起来后，就开始执行发送回调（）的这个方法，而且只执行一次，所以必须是定时任务，定时发送0步骤定义的队列，如果为空就跳出方法 return,不执行下面的程序

2.定义业务要发送的回调callback:即业务层做完一件事，需要发送时调用它，

方法发送的时候，要把回调信息保存在0步骤定义的队列变量中，将会在下次定时发出

# [34.componet](mailto:34.@componet)

表示该类为可加载到容器的class

\*系统启动后，开始扫描所有被注解的类成bean（已经单例初始化）注入系统中，待其他程序加载到上下文，@autowire，系统启动后，都已经加载到上下文了其实，即bean可用的状态

# [34.try](mailto:34.@componet) catch

**我们编译运行程序出错的时候，编译器就会抛出异常。抛出异常要比终止程序灵活许多，这是因为**

**的Java提供了一个“捕获”异常的的处理器（处理器）对异常情况进行处理。**

**如果没有提供处理器机制，程序就会终止，并在控制台上打印一条信息，给出异常的类型.L**

**比如：使用了NULL引用或者是数组越界等。**

**异常有两种类型：未检查异常和已检查异常对于已检查异常，处理器器将会检查是否提供了处理器。**

**然而有许多の异常，如：访问null引用，都属于未检查异常。编译器不会查看是否为这些错误提供了**

**处理器。毕竟，应该用严谨的态度来对待写代码，依次避免这些错误的发生，而不是将精力花在编写**

**异常处理器上。**

**废话少说，show coder：**

**/ \* try catch：自己处理异常**

**\* try {**

**\*可能出现异常的代码（可以自定义A异常类，普通类继承异常类例如：RuntimeException**

**（api接口（异常字段用枚举来表示）的校验）异常：即符合什么条件下，例：变量==空；抛出异常；用 throw e ，然后cattch会捕获到A类的异常(例子)）**

**\*} catch（异常类名A e）{**

**\*如果出现了异常类A类型的异常，那么执行该代码**

**\*} ...（catch可以有多个）**

**\* finally {**

**\*最终肯定必须要执行的代码（例如释放资源的代码）**

**\*}**

**\*代码执行的顺序：**

**\* 1.try内的代码从出现异常的那一行开始，中断执行**

**\* 2.执行对应的catch块内的代码**

**\* 3.继续执行try catch结构之后的代码**

**\*注意点：**

**\* 1.如果catch内的异常类存在子父类的关系，那么子类应该在前，父类在后**

**\* 2。如果最后中有返回语句，那么最后返回的结果肯定以最终中的返回值为准**

**\* 3。如果最后语句中有回报，那么没有被处理的异常将会被吞掉**

**\*重写的注意点：**

**\* 1.儿子不能比父亲的本事大**

**\* 2.儿子要比父亲开放**

**\* 3.儿子不能比父亲惹更大的麻烦（子类的异常的类型不能是父类的异常的父类型）**

**\*异常类Api：**

**Rms Api异常:**

APIException 继承RuntimeException，其中

public class APIException extends RuntimeException {  
 private static final long *serialVersionUID* = 1L;  
  
 */\*\*  
 \* 错误码  
 \*/* private int code;  
 */\*\*  
 \* 错误描述  
 \*/* private String msg;  
 */\*\*  
 \* 错误描述中占位符所对应的参数  
 \*/* private transient Object[] args;  
  
  
 public APIException(APIResult result) {  
 super(result.getMsg());  
 this.code = result.getCode();  
 this.msg = result.getMsg();  
 }  
  
 public APIException(APIResult result, Object... args) {  
 super(result.getMsg());  
 this.code = result.getCode();  
 this.msg = result.getMsg();  
 this.args = args;  
 }  
  
 public APIException(int code, String msg) {  
 super(msg);  
 this.code = code;  
 this.msg = msg;  
 }  
  
 public APIException(int code, String msg, Object... args) {  
 super(msg);  
 this.code = code;  
 this.msg = msg;  
 this.args = args;  
 }  
  
 */\*\*  
 \** ***@param*** *obj  
 \** ***@param*** *message  
 \** ***@param*** <*T*>  
 *\** ***@return*** *\*/* public static <T> T requireNonNull(T obj, String message) {  
 if (obj == null) {  
 throw new APIException(APIResult.*ILLEGAL\_ARGUMENT\_NULL*.getCode(), message);  
 }  
 return obj;  
 }  
  
 public static <T> T requireNonNull(T obj, APIResult result) {  
 if (obj == null) {  
 throw new APIException(result);  
 }  
 return obj;  
 }  
  
 public static <T> Collection<T> requireNonEmpty(Collection<T> obj, APIResult result) {  
 if (obj == null || obj.isEmpty()) {  
 throw new APIException(result);  
 }  
 return obj;  
 }  
  
 */\*\*  
 \* 如果指定对象为null，则抛出一个异常  
 \*  
 \** ***@param*** *obj  
 \** ***@param*** *result  
 \** ***@param*** *args  
 \** ***@param*** <*T*>  
 *\** ***@return*** *\*/* public static <T> T requireNonNull(T obj, APIResult result, Object... args) {  
 if (obj == null) {  
 throw APIException.*builder*()  
 .code(result.getCode())  
 .msg(result.getMsg())  
 .args(args)  
 .build();  
 }  
 return obj;  
 }  
  
 */\*\*  
 \* 如果指定集合类型为空，则抛出一个API异常  
 \*  
 \** ***@param*** *obj  
 \** ***@param*** *result  
 \** ***@param*** *args  
 \** ***@param*** <*T*>  
 *\** ***@return*** *\*/* public static <T> Collection<T> requireNonEmpty(Collection<T> obj, APIResult result, Object... args) {  
 if (obj == null || obj.isEmpty()) {  
 throw APIException.*builder*()  
 .code(result.getCode())  
 .msg(result.getMsg())  
 .args(args)  
 .build();  
 }  
 return obj;  
 }  
  
 */\*\*  
 \* 抛出一个API异常  
 \*  
 \** ***@param*** *result  
 \** ***@param*** *args  
 \*/* public static void throwAPIException(APIResult result, Object... args) {  
 throw APIException.*builder*()  
 .code(result.getCode())  
 .msg(result.getMsg())  
 .args(args)  
 .build();  
 }  
  
 @Override  
 public String toString() {  
 return "APIException{"  
 + "code=" + code  
 + ", msg='" + msg + '\''  
 + ", args=" + Arrays.*toString*(args) + '}';  
 }  
}

业务层具体抛出此类异常APIException 调用方法verifyMsgBasic（）

private void verifyMsgBasic(JsonNode jsonNode) {  
 // parse to java bean  
 if (!jsonNode.has("id") || StringUtil.*isEmpty*(jsonNode.get("id").asText())) {  
 throw new APIException(APIResult.*REQUEST\_MSG\_ID\_NULL*);  
 }  
  
 if (!jsonNode.has("msgType") || StringUtil.*isEmpty*(jsonNode.get("msgType").asText())) {  
 throw new APIException(APIResult.*REQUEST\_MSG\_TYPE\_NULL*);  
 } else if (msgFactory.getMsgType(jsonNode.get("msgType").asText()) == null) {  
 throw new APIException(APIResult.*REQUEST\_MSG\_TYPE\_NOT\_SUPPORTED*, jsonNode.get("msgType").asText());  
 }  
}

调用此方法并用try catch 捕获到并放到响应消息body中

private AbstractMsg parseToMsg(String content, APIProtocolType protoType, String remoteAddr) {  
 try {  
 // parse to json node  
 JsonNode jsonNode = JSONUtil.*jsonToNode*(content);  
  
 // verify message basic info  
 verifyMsgBasic(jsonNode);  
  
 // parse to java bean  
 return JSONUtil.*nodeToObj*(jsonNode, msgFactory.getMsgType(jsonNode.get("msgType").asText()));  
  
 } catch (Exception e) {

msgHandler.saveUnParsableReuestMsg(content, protoType.name(), remoteAddr);  
 *log*.error("parse msg failed! protoType:{}, remoteAddr:{}, msg:{}", protoType, remoteAddr, content);

//将异常响应定义为异常，并放入响应body中  
 return msgFactory.newUnparsableMsg(content, e);  
 }  
}

自定义异常类需要用的api结果集用枚举列

*API结果枚举定义  
 \*  
 \** ***@author*** *刘俊  
 \** ***@since*** *2.5.0  
 \*/*public enum APIResult {  
  
 */\*\*  
 \* 成功  
 \*\*/  
 OK*(0, APII18nConstant.*API\_RESULT\_OK*),  
 *OK\_PART*(0, APII18nConstant.*API\_RESULT\_OK\_PART*),  
  
 */\*\*  
 \* 异常或其它未知错误  
 \*\*/  
 ERROR*(100, APII18nConstant.*API\_RESULT\_ERROR*),  
 */\*\*  
 \* 异常或其它未知错误  
 \*\*/  
 NOT\_SUPPORT*(101, APII18nConstant.*API\_RESULT\_NOT\_SUPPORT*),  
  
 */\*\*  
 \* RMS正在初始化  
 \*/  
 RMS\_INITIALIZING*(110, APII18nConstant.*API\_RESULT\_NOT\_INITIALIZING*),  
  
 */\*\*  
 \* 操作失败  
 \*\*/  
 OPERATION\_FAILED*(121, APII18nConstant.*API\_RESULT\_OPERATION\_FAILED*),  
  
 */\*\*  
 \* 无效的参数  
 \*\*/* APII18nConstant.*MONITOR\_WEB\_COMMON\_OBJECT\_ALREADY\_EXISTS*);  
  
 private int code;  
 private String msg;  
  
 APIResult(int code, String msg) {  
 this.code = code;  
 this.msg = msg;  
 }  
  
 public int getCode(  
 ) {  
 return code;  
 }  
  
 public String getMsg(  
 ) {  
 return msg;  
 }  
  
}

# 35.@Builder(花式)

**\*lombook的一个注解，用于注解在类上，可以花式创建对象，**

....鲍勃是你的叔叔：用于创建对象的无懈可击的花式裤子！  
@Builder 在lombok v0.12.0中作为实验特征介绍。  
@Builder获得了@Singular支持，并lombok从lombok v1.16.0 升级到主程序包。  
@Builder与@Singular增加，因为龙目岛v1.16.8一个明确的方法。  
@Builder.Default 功能已在lombok v1.16.16中添加。

## Overview

该@Builder标注生产络合剂的API为你的类。

@Builder 允许您使用以下代码自动生成使您的类可实例化所需的代码：  
Person.builder().name("Adam Savage").city("San Francisco").job("Mythbusters").job("Unchained Reaction").build();

\*集合

@Builder可以为集合参数/字段生成所谓的“奇异”方法。它们采用1个元素而不是整个列表，并将该元素添加到列表中。例如：Person.builder().job("Mythbusters").job("Unchained Reaction").build();将导致该List<String> jobs字段中包含2个字符串。要获得此行为，需要使用注释字段/参数@Singular。

### @Builder.Default

如果在构建会话期间从未设置某个字段/参数，则它始终为0 / null/ false。如果您已经放置@Builder了一个类（而不是方法或构造函数），则可以直接在该字段上指定默认值，并使用以下内容对该字段进行注释@Builder.Default：  
@Builder.Default private final long created = System.currentTimeMillis();

\*\*当子类继承父类后想使用@builder,父子类都应用@superBuilder(lombok版本：1.18.8以上的版本)

# 35. Primary 注解

\*用在componet标注的类上，还可以标在配置类的@bean标注的注册bean的方法上();

对于同一个接口people 有多个实现类时，比如men ,women,当其他类引入此peple bean时，例如student类用@autowire引入时，需要在men类(或women)类上加入注解@primary表示默认加载的是men 的bean,如下：

@componet(代表把这个类注册成bean)

@primary(注册成bean时把他作为默认被调用的bean)

Public men implements people{

@overwrite

Say(){

System.out.println(我是男人)

}

}

Public women implements people{

@overwrite

Say(){

System.out.println(我是女人)

}

}

Public class sduent{

@autowired

People people;

People.say();

}

输出结果：我是男人；

。

# 36. Autowired注解

@Autowired(required = false),没有注册该bean也不会报错

**@qualifier**

**当接口例People有多个实现类时，实现类用@componet，@services@bean修饰默认没有bean名字要加上value属性@componet(value=”student”)**

**其他类注入时需要用@qualifier过滤**

**@autowired**

**@qualifier（value=”suudent”）,**

**People peple;**

# 37.二维数组

定义格式

\* a 第一种定义格式:

\* int[][] arr = new int[3][4];// arr里面包含3个数组 每个数组里面有四个元素(3行4列，一行一个数组)

\* 上面的代码相当于定义了一个3\*4的二维数组，即二维数组的长度为3，二维数组中的每个元素又是一个长度为4的数组

\* b 第二种定义格式

\* int[][] arr = new int[3][];

\* 第二种方式和第一种类似，只是数组中每个元素的长度不确定

\* c 第三种定义格式

\* int[][] arr = {{1,2},{3,4,5,6},{7,8,9}};

\* 二维数组中定义了三个元素，这三个元素都是数组，分别为{1,2}、{3,4,5,6}、{7,8,9}

二维数组的遍历

public class ArrayArrayDemo1{

public static void main(String[] args){

int[][] arr={{12,58},{56,78,41},{0}};

//System.out.println(arr[1][1]);

for(int i=0;i<arr.length;i++){

for(int j=0;j<arr[i].length;j++){

System.out.print(arr[i][j]+" ");

}

System.out.println();

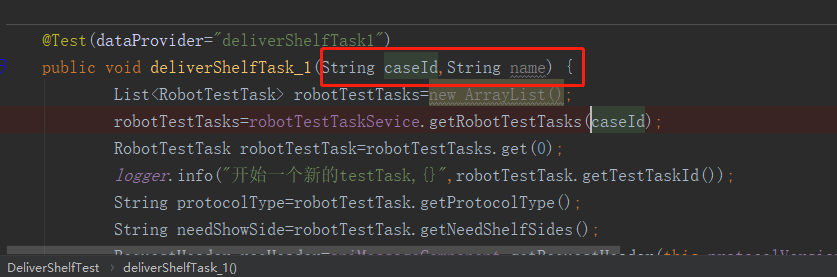
}

}

}

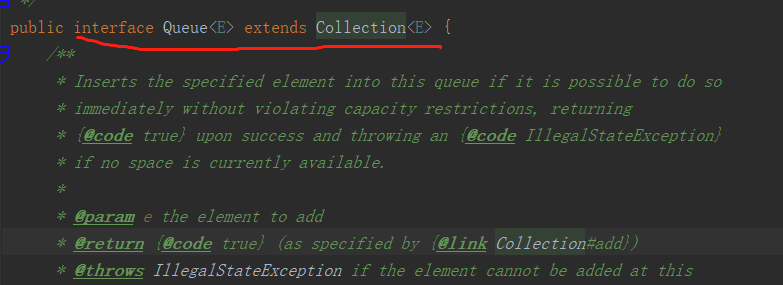
@DataProvider(name="deliverShelfTask1")  
 public Object[][] getDeliverShelf(){  
 List<String> testCaseIdList=robotTestTaskSevice.getCaseIds("deliverShelfTask1");  
 int size = testCaseIdList.size();  
 Object[][] Data = new Object[size][];  
 for (int i=0;i<size;i++){  
 String caseId=testCaseIdList.get(i);  
 Data[i]=new Object[]{caseId,name};  
 }  
 return Data;  
 }  
}

DataProvider数据源为二维数组，调用次数为二维数组data【】【】的长度，即每次调用一个data【i】,参数就是data【i】的元素data【i】｛caseid,name｝参数顺序必须一致，如下图：

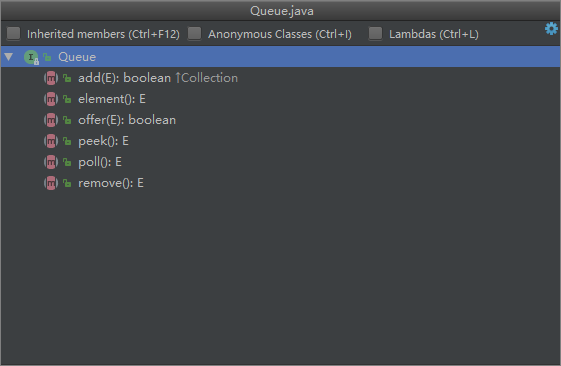


# 38．队列

Queue定义



属性方法



获得初始化对象

private final transient Queue<CallbackEntry> clientCallbackQueue = new LinkedBlockingQueue<>();

\*因为LinkedBlockingQueue实现了Queue接口，所以可以这么初始化，LinkedBlockingQueue是链式存储；

下面讲解queue

　先我们要知道使用队列的目的是什么？一般情况下，如果是一些及时消息的处理，并且处理时间很短的情况下是不需要使用队列的，直接阻塞式的方法调用就可以了。但是，如果在消息处理的时候特别费时间，这个时候如果有新的消息来了，就只能处于阻塞状态，造成用户等待。这个时候在项目中引入队列是十分有必要的。当我们接受到消息后，先把消息放到队列中，然后再用新的线程进行处理，这个时候就不会有消息的阻塞了。下面就跟大家介绍两种队列的使用，一种是基于内存的，一种是基于数据库的。

     首先，我们来看看基于内存的队列。在Java的并发包中已经提供了BlockingQueue的实现，比较常用的有ArrayBlockingQueue和LinkedBlockingQueue，前者是以数组的形式存储，后者是以Node节点的链表形式存储。至于数组和链表的区别这里就不多说了。

BlockingQueue 队列常用的操作方法:

      1.往队列中添加元素: **add(), put(), offer()**

      2.从队列中取出或者删除元素: **remove() element()  peek()   poll()  take()**

每个方法的说明如下：

**offer()**方法往队列添加元素如果队列已满直接返回false,队列未满则直接插入并返回true;

**add()**方法是对offer()方法的简单封装.如果队列已满,抛出异常new IllegalStateException("Queue full");

**put()**方法往队列里插入元素,如果队列已经满,则会一直等待直到队列为空插入新元素,或者线程被中断抛出异常.

**remove()**方法直接删除队头的元素:

**peek()**方法直接取出队头的元素,并不删除.

**element()**方法对peek方法进行简单封装,如果队头元素存在则取出并不删除,如果不存在抛出异常NoSuchElementException()

**poll()**方法取出并删除队头的元素,当队列为空,返回null;

**take()**方法取出并删除队头的元素,当队列为空,则会一直等待直到队列有新元素可以取出,或者线程被中断抛出异常

　　offer()方法一般跟pool()方法相对应, put()方法一般跟take()方法相对应.日常开发过程中offer()与pool()方法用的相对比较频繁.

下面用一个例子来看看是怎么使用的。

# 39．instanceof

1. Object instanceof Class

注：object必须是引用类型，即对象，不能基本数据类型 int i=1 ; I instancof Integer;是错误的， 必须是引用对象，即必须封箱，可以是接口的实现

Class：必须是父类，接口，引用对象的父类，（基本数据与引用类型见40）

例如下面：object是最大的引用对象，所以可以使用val instanceof CharSequence

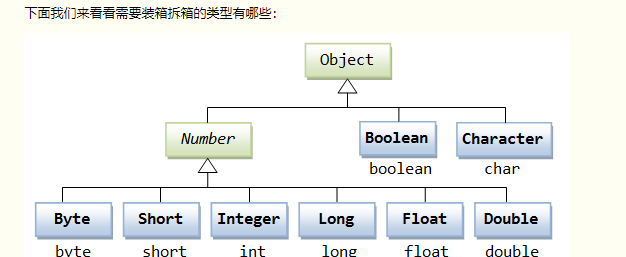
Object val = map.get(key);//自动封箱成引用对象，  
if (val instanceof CharSequence) {  
 if ("".equals(val.toString().trim())) {  
 return null;  
 }  
 return Float.*parseFloat*(val.toString());  
}

CharSequence为String得父类,可读可写有序字符串，string只能读

注意：1、instanceof用于判断一个对象是否是某个类的实例。

2、在对象向下转型（父类转成子类）之前最好使用instanceof关键字进行验证。确定属于哪个子类，否则会发生ClassCastException

# 40.[java基本类型与引用类型](https://www.cnblogs.com/fengda/p/10125970.html)



**Object是最大的封箱**

目录  
java基本类型与引用类型  
目录  
一基本数据类型  
二引用类型  
三基本类型与引用类型的区别  
默认值  
内存分配  
自动装箱自动拆箱  
自动装箱拆箱带来的问题  
程序的性能  
空指针异常  
对象相等比较时  
一、基本数据类型  
java中一共分为8种基本数据类型：byte、short、int、long、float、double、char、boolean,其中byte、short、int、long是整型。float、double是浮点型,char是字符型,boolean是布尔型。

二、引用类型  
java为每种基本类型都提供了对应的封装类型，分别为：Byte、Short、Integer、Long、Float、Double、Character、Boolean。引用类型是一种对象类型,它的值是指向内存空间的引用，就是地址。

三、基本类型与引用类型的区别  
1.默认值  
整型byte、short、int、long的默认值都为0,浮点型float、double的默认值为0.0,boolean默认值为false,char默认值为空。对应的包装类型默认值都为null。

2.内存分配  
基本数据类型的变量是存储在栈内存中，而引用类型变量存储在栈内存中，保存的是实际对象在堆内存中的地址，实际对象中保存这内容。

3.自动装箱、自动拆箱  
Java从jdk1.5开始引入自动装箱和拆箱,使得基本数据类型与引用类型之间相互转换变得简单。

自动装箱: java自动将原始类型转化为引用类型的过程，自动装箱时编译器会调用valueOf方法,将原始类型转化为对象类型。即都调用Integer.valueof()返回Integer类型的对象，注意可能有区间范围返回同一个对象，所以会相等

自动拆箱: java自动将引用类型转化为原始类型的过程，自动拆箱时编译器会调用intValue(),doubleValue()这类的方法将对象转换成原始类型值。即Integer.inValue（）返回对应的值

自动装箱主要发生在两种情况：一种是赋值时，一种是方法调用时。   
a.赋值

Integer a = 3; //自动装箱，等同于Integer.valueof(3)  
int b = a; //自动拆箱,等同于int b=a.intValue()=3  
b.方法调用

public Integer query(Integer a){  
return a;  
}  
query(3); //自动装箱//因为参数类型为引用类型，所以3是int必须自动装箱  
int result = query(3); //自动拆箱，因为定义result为int（基础类型）所以必须自动拆箱；

**拆箱原则**：1、当一个基础数据类型与封装类进行==、+、-、\*、/运算时，会将封装类进行拆箱，对基础数据类型进行运算

其他拆封根据赋值的类型或方法调用的参数类型决定，如Integer a = 3; query(3)都是封箱，

int result = query(3); int b = a都是拆箱  
4.自动装箱、拆箱带来的问题  
1.程序的性能  
由于装箱会隐式地创建对象创建，因此千万不要在一个循环中进行自动装箱的操作，下面就是一个循环中进行自动装箱的例子，会额外创建多余的对象,增加GC的压力,影响程序的性能：

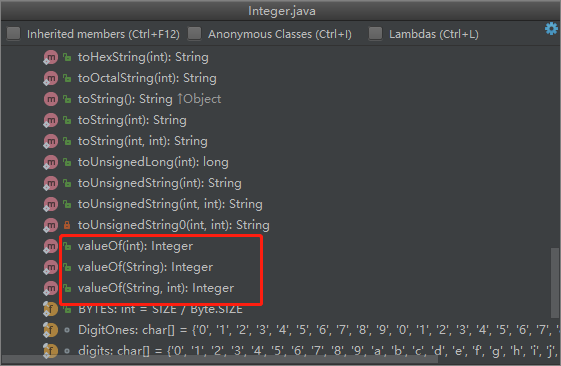
Integer sum = 0;  
for(int i=0; i<1000; i++){  
sum+=i;  
}  
2.空指针异常  
注意拆箱过程中可能产生的空指针异常，一个简单的例子：

Object obj = null;  
int i = (Integer)obj;  
3.对象相等比较时  
先来看一个常见的例子：

Integer a = 120;//这个进行了装箱，初始化了对象，默认调用了Intger类的valueof()方法进行了初始化，  
int b= 120;  
Integer c = 120;  
Integer d = new Integer(120);//d是个对象，要想调用值得调用

解释下引用对象Integer d = new Integer(120)

public final class Integer extends Number implements Comparable<Integer> {



System.out.println(a == b); //true t1  
System.out.println(a == c); //true t2  
System.out.println(a == d); //false t3

Integer e = 128;  
Integer f = 128;  
System.out.println(e == f); //false t4  
返回结果是不是出乎大家的意料,解释一下每种结果的原因：   
我们先反编译一下生成字节码：

Integer a = Integer.valueOf(120);  
int b = 120;  
Integer c = Integer.valueOf(120);  
Integer d = new Integer(120);  
System.out.println(a.intValue() == b);  
System.out.println(a == c);  
System.out.println(a == d);

Integer e = Integer.valueOf(127);  
Integer f = Integer.valueOf(127);  
System.out.println(e == f);

Integer e1 = Integer.valueOf(128);  
Integer f1 = Integer.valueOf(128);  
System.out.println(e1 == f1);  
可以看到变量a、c在初始化的时候编译器调用了valueOf进行自动装箱，在a==b时对变量a调用了intValue()方法进行了自动拆箱操作，这就很好解释t1~t4的结果了。A是个数值，120，d是个对象120

t1产生的原因是编译器编译时会调用intValue()自动的将a进行了拆箱，结果肯定是true;   
t2跟t4的结果比较难理解：这是因为初始化时，编译器会调用装箱类的valueOf()方法,查看jdk的源码：

public static Integer valueOf(int i) {  
assert IntegerCache.high >= 127;  
if (i >= IntegerCache.low && i <= IntegerCache.high)  
return IntegerCache.cache[i + (-IntegerCache.low)];  
return new Integer(i);  
}  
  
发现jdk对-128~127之间的值做了缓存，对于-128~127之间的值会取缓存中的引用,通过缓存经常请求的值而显著提高空间和时间性能。   
这就能解释t2结果返回true，而t4由于128不在缓存区间内，编译器调用valueOf方法会重新创建新的对象，两个不同的对象返回false。

t3结果无论如何都不会相等的，因为new Integer(120)构造器会创建新的对象。

Byte、Short、Integer、Long、Char这几个装箱类的valueOf()方法都会做缓存，而Float、Double则不会，原因也很简单，因为byte、Short、integer、long、char在某个范围内的整数个数是有限的，但是float、double这两个浮点数却不是

# 40.[Char(Character)基本类型与String](https://www.cnblogs.com/fengda/p/10125970.html)（不是基本类型）

1.Char 是基本数据类型，字符类型，引用类型Character，占用2byte(因为汉字太多必须用两个字字节)。它是组成分隔String字符串的最小单位

定义：用单引号，里边只能有一个字符，字母或者数字，或汉字，进制char a=‘a’

用16进制 0x开头，（0开头8进制）因为1111对应16进制F，两个字节最大也就FFFF，所以 Char b= 0xFFFF；

1. String类，字符串实现了CharSequence接口

|  |
| --- |
| String greeting = "Hello world!"; |

在代码中遇到字符串常量时，这里的值是"Hello world!"，编译器会使用该值创建一个String对象。

和其它对象一样，可以使用关键字和构造方法来创建String对象。

String类有11种构造方法，这些方法提供不同的参数来初始化字符串，比如提供一个字符数组参数:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class StringDemo{       public static void main(String args[]){        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};        String helloString = new String(helloArray);        System.out.println( helloString );     }  } |

以上实例编译运行结果如下：

|  |  |
| --- | --- |
| 1 | hello. |

**注意:**String类是不可改变的，所以你一旦创建了String对象，那它的值就无法改变了。 如果需要对字符串做很多修改，那么应该选择使用StringBuffer & StringBuilder 类。他们都实现了父接口CharSequence

所以在数据类型转换时可以这么判断object instancof CharSequence

# 41.[接口设计之封装map](https://www.cnblogs.com/fengda/p/10125970.html)

可以将接口的key与value值封装成map,如下

private final Map<String, Object> map;

public MapEntity() {  
 this.map = new LinkedHashMap<>();  
}

他的值是object超类，可以是任自己事先定义好自己需要的类型，普通的java类也可以，然后封装getstring,getboolen方法获取相应类型的数据类型，还可以与json封装直接转换成json，互相转化，以下为获取数据get方法

public Integer getInteger(String key) {  
 Objects.*requireNonNull*(key);//如果key为空抛出空指针，否则还是返回原值，判断key合法性  
  
 Object val = map.get(key);//进行装箱，比如是个3装成Intenger

//判断val是否是CharSequence 字符串的子类实例  
 if (val instanceof CharSequence) {  
 if ("".equals(val.toString().trim())) {  
 return null;  
 }

//如果是，并不为空返回int  
 return Integer.*parseInt*(val.toString());  
 }  
如果不是string,肯定是numbenr类型子类对象，number包括所有的基础数据类型，除了字符类型，强制转化为Number类型，

Number number = (Number) val;//多态：父类引用指向子类，调用方法时，如果父方法不在子类里，报错，如果存在则调用子类的方法。Numbenr抽象类，本身抽象方法（没有方法体）是多态的表现，因为子类必须实现抽象类抽象方法，抽象类不能使用new 创建对象，可以指向子类对象，多态  
 if (number == null) {  
 return null;  
 } else if (number instanceof Integer) {  
 return (Integer) number; // 避免不必要的拆箱和装箱  
 } else {

//如果是其他比如float类型，调用number的intValue方法，拆箱，返回int,以下调用的是子类的float得intValue方法  
 return number.intValue();  
 }  
}

# 42.接口和抽象类的联系和区别

 一，简单总结

 　　1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

　　2、抽象类要被子类继承，接口要被类实现。

　　3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

　　4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

　　5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现  　　接 口方法，那么该类也只能为抽象类。

　　6、抽象方法只能申明，不能实现，接口是设计的结果 ，抽象类是重构的结果

　　7、抽象类里可以没有抽象方法

　　8、如果一个类里有抽象方法，那么这个类只能是抽象类

　　9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

　　10、接口可继承接口，并可多继承接口，但类只能单根继承。

  二，标准作答

　　相同点

* 抽象类和接口均包含抽象方法，类必须实现所有的抽象方法，否则是抽象类
* 抽象类和接口都不能实例化，他们位于继承树的顶端，被其他类继承和实现

两者的区别主要体现在两方面：语法方面和设计理念方面

语法方面的区别是比较低层次的，非本质的，主要表现在：

* 接口中只能定义全局静态常量，不能定义变量。抽象类中可以定义常量和变量。
* 接口中所有的方法都是全局抽象方法。抽象类中可以有0个、1个或多个，甚至全部都是抽象方法。
* 抽象类中可以有构造方法，但不能用来实例化，而在子类实例化时执行，完成属于抽象类的初始化操作。接口中不能定义构造方法。
* 一个类只能有一个直接父类（可以是抽象类），但可以充实实现多个接口。一个类使用extends来继承抽象类，使用implements来实现接口。

二者的主要区别在设计理念上，决定了某些情况下使用抽象类还是接口。

* 抽象类体现了一种继承关系，目的是复用代码，抽象类定义了各个子类的相同代码，可认为父类是一个实现了部分功能的“中间产品”，而子类是“最终产品”。父类和子类间须存在“is-a”关系，即父类和子类在概念本质上是相同的。

接口并不要求实现类和接口在概念本质上一致，仅仅是实现了接口定义的约定或能力而已。接口定义了“做什么”，实现类负责完成“怎么做”，体现了功能（规范）和实现分离的原则。接口和实现之间可以认为是一种“has-a的关系

# 43.变量的访问范围

**Private:被它修饰的属性或方法只能在当前类里可以访问，一般定义属性，如果访问，通过共有定义的方法。**

关键词表示在其他类里不能通过对象点属性访问，只能通过定义其他public方法，改方法访问私有变量，间接的访问

如：age不能被其他地方的类直接访问，只能通过共有方法setAge

Public student{

Private int age;

@autowired

Private Parent parent //只在当前类调用，避免

Public void setAge(int age)

This.age=age

}

**public:其他任意类都可以直接访问;一般只用来定义方法，类**

**protected:** 子类，同package下类可以访问

# 44.单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。Springboot @componet,等标默认表示为bean得类bean都是单例模式；在系统启动时都创建好了，注册进来了@autowire只是把实例对象加载进来了，其实都是一个对象

@Component注解默认实例化的对象是单例，如果想声明成多例对象可以使用@Scope("prototype")

@Repository默认单例

@Service默认单例

@Controller默认单例

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

**注意：**

* 1、单例类只能有一个实例。
* 2、单例类必须自己创建自己的唯一实例。
* 3、单例类必须给所有其他对象提供这一实例。

## 介绍

**意图：**保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**主要解决：**一个全局使用的类频繁地创建与销毁。(系统资源类，创建一遍就行，)

**何时使用：**当您想控制实例数目为1，公共资源，节省系统资源的时候。，如果像nettyserver一样，有http,soiket多个配置，肯定不用@compnet标注，因为要有多个实例，对应各自不同的变量值

**如何解决：**判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

**关键代码：**构造函数是私有的。

**应用实例：**

* 1、一个班级只有一个班主任。
* 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。
* 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

**优点：**

* 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。
* 2、避免对资源的多重占用（比如写文件操作）。

**缺点：**没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

**使用场景：**

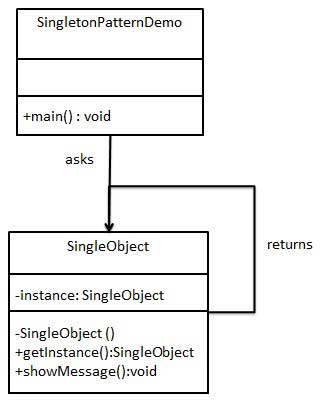
* 1、要求生产唯一序列号。
* 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。
* 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

**注意事项：**getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

## 实现

我们将创建一个 *SingleObject* 类。*SingleObject* 类有它的私有构造函数和本身的一个静态实例。

*SingleObject* 类提供了一个静态方法，供外界获取它的静态实例。*SingletonPatternDemo*，我们的演示类使用 *SingleObject* 类来获取 *SingleObject*对象。



创建一个 Singleton 类。



## 单例模式的几种实现方式

单例模式的实现有多种方式，如下所示：

### 1、懒汉式，线程不安全

**是否 Lazy 初始化：**是

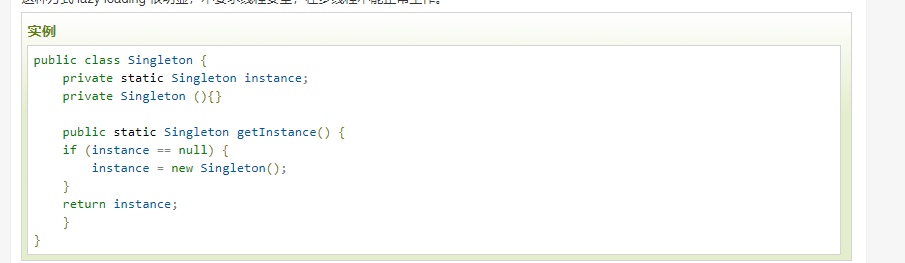
**是否多线程安全：**否

**实现难度：**易

**描述：**这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。  
这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

## 实例

public class Singleton { private static Singleton instance; private Singleton (){} public static Singleton getInstance() { if (instance == null) { instance = new Singleton(); } return instance; } }



**接下来介绍的几种实现方式都支持多线程，但是在性能上有所差异。**

### 2、懒汉式，线程安全

**是否 Lazy 初始化：**是

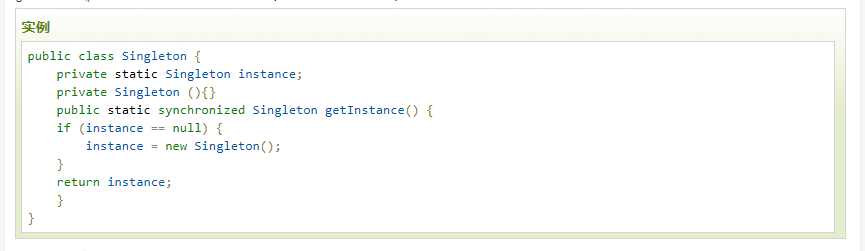
**是否多线程安全：**是

**实现难度：**易

**描述：**这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。  
优点：第一次调用才初始化，避免内存浪费。  
缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。  
getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

## 实例

public class Singleton { private static Singleton instance; private Singleton (){} public static synchronized Singleton getInstance() { if (instance == null) { instance = new Singleton(); } return instance; } }



### 3、饿汉式

**是否 Lazy 初始化：**否

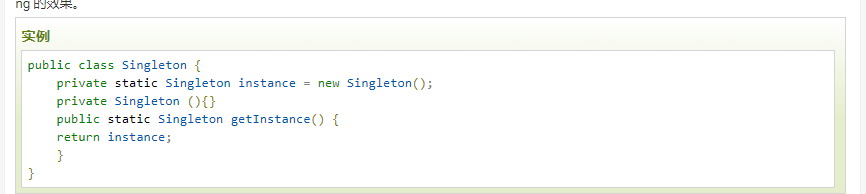
**是否多线程安全：**是

**实现难度：**易

**描述：**这种方式比较常用，但容易产生垃圾对象。  
优点：没有加锁，执行效率会提高。  
缺点：类加载时就初始化，浪费内存。  
它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

## 实例

public class Singleton { private static Singleton instance = new Singleton(); private Singleton (){} public static Singleton getInstance() { return instance; } }



### 4、双检锁/双重校验锁（DCL，即 double-checked locking）

**JDK 版本：**JDK1.5 起

**是否 Lazy 初始化：**是

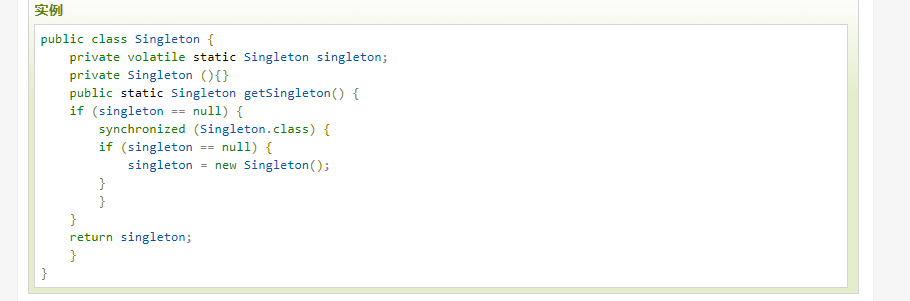
**是否多线程安全：**是

**实现难度：**较复杂

**描述：**这种方式采用双锁机制，安全且在多线程情况下能保持高性能。  
getInstance() 的性能对应用程序很关键。

## 实例

public class Singleton { private volatile static Singleton singleton; private Singleton (){} public static Singleton getSingleton() { if (singleton == null) { synchronized (Singleton.class) { if (singleton == null) { singleton = new Singleton(); } } } return singleton; } }



### 5、登记式/静态内部类

**是否 Lazy 初始化：**是

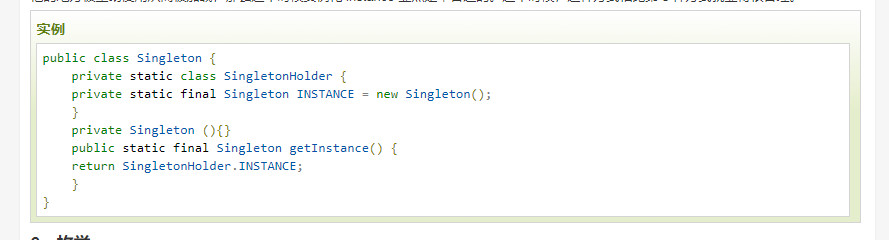
**是否多线程安全：**是

**实现难度：**一般

**描述：**这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。  
这种方式同样利用了 classloader 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。

## 实例

public class Singleton { private static class SingletonHolder { private static final Singleton INSTANCE = new Singleton(); } private Singleton (){} public static final Singleton getInstance() { return SingletonHolder.INSTANCE; } }



### 6、枚举

**JDK 版本：**JDK1.5 起

**是否 Lazy 初始化：**否

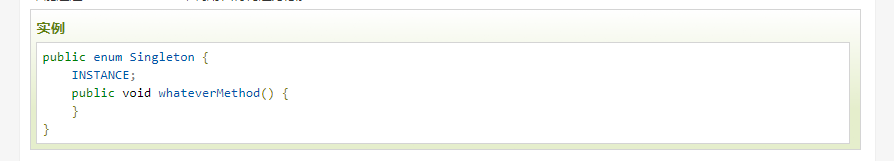
**是否多线程安全：**是

**实现难度：**易

**描述：**这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。  
这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。  
不能通过 reflection attack 来调用私有构造方法。

## 实例

public enum Singleton { INSTANCE; public void whateverMethod() { } }



**经验之谈：**一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

# 45.抽象类实现接口

设计思想：当你自己写的类**想用接口中个别方法的时候**（注意不是所有的方法），那么你就可以**用一个抽象类先实现这个接口**（方法体中为空），然后再用你的类继承这个抽象类，这样就可以达到你的目的了，如果你直接用类实现接口，那是所有方法都必须实现的；

注： java.swing.event包中运用的非常多，里面一般以Adapter为后缀的都是抽象类，它们都实现了特定的事件

例：有一个接口Window，有三个方法，draw(),putColor(),setPosition()三个方法，程序员在设计页面时只关注对其进行位置定位，只需实现setPosition()（画图draw()和着色putColor()则由美工实现），所以他只需要实现setPosition()方法，而其余两个不用实现。则设计如下：

//接口

interface Window{

public void draw();

public void putColor();

public void setPosition();

}

/\*\*

\*抽象类，不实现需要的方法。而实现的不需要的方法的方法体设为空，，已经实现的无需在子类中实现了。但抽象类+具体实现类总共实现的方法必须是所有接口方法

\*/

abstract class DesignedPage implements Window{

public void draw(){}

public void putColor(){}

}

/\*\*

\* 具体实现类，继承抽象类，实现特定的方法setPosition，该类中只有需要的方法，

\*/

public class DesingedPageA extends DesignedPage{

public void setPosition(){

//set the window position

}

}

# 46.内部匿名类直接new接口

## 1.内部匿名实现接口

可以在别的类里直接通过内部匿名类的方式(new)实现接口创建个实例

java中的匿名类有一个倍儿神奇的用法，见下面代码示例:

1 package contract;

2

3 public interface ISay {

4 void sayHello();

5 }

上面是一个简单的接口，下面是如何使用：只能在方法中定义

[复制代码](javascript:void(0);)

1 package jimmy;

2 import contract.ISay;

3 public class Program {

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

//以下new （）后的｛｝里直接重写实现接口方法，就是内部匿名类创建了接口的实现子类say

5 ISay say = new ISay() {

6 public void sayHello() {

7 System.out.println("Hello java!");

8 }

9 };

Say.say()

10 }

11 }



输出：hello java

## 2.内部类

**在java语言中，可以把一个类定义到另外一个类的内部，在类里面的这个类就叫内部类，外面的类就叫外部类。**

**在这情况下，这个内部类可以看做外部类的一个成员。**

**内部类好处**

**1.隐藏你不想让别人知道的操作，也即封装性。**

**public interface Contents {**

**int value();**

**}**

**public interface Destination {**

**String readLabel();**

**}**

**public class Goods {**

**private class Content implements Contents {**

**private int i = 11;**

**public int value() {**

**return i;**

**}**

**}**

**protected class GDestination implements Destination {**

**private String label;**

**private GDestination(String whereTo) {**

**label = whereTo;**

**}**

**public String readLabel() {**

**return label;**

**}**

**}**

**public Destination dest(String s) {**

**return new GDestination(s);**

**}**

**public Contents cont() {**

**return new Content();**

**}**

**}**

**class TestGoods {**

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

**Goods p = new Goods();**

**Contents c = p.cont();**

**Destination d = p.dest("Beijing");**

**}**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28**

**29**

**30**

**31**

**32**

**33**

**34**

**35**

**36**

**37**

**38**

**39**

**40**

**41**

**42**

**43**

**44**

**45**

**46**

**在这个例子里类Content和GDestination被定义在了类Goods内部，并且分别有着protected和private修饰符来控制访问级别。在后面的main方法里，直接用 Contents c和Destination d进行操作，你甚至连这两个内部类的名字都没有看见！这样，内部类的第一个好处就体现出来了 隐藏你不想让别人知道的操作，也即封装性。同时，我们也发现了在外部类作用范围之外得到内部类对象的第一个方法，那就是利用其外部类的方法创建并返回。**

**2.一个内部类对象可以访问创建它的外部类对象的内容，甚至包括私有变量！**

**public class Goods {**

**private valueRate = 2;**

**private class Content implements Contents {**

**private int i = 11 \* valueRate;**

**public int value() {**

**return i;**

**}**

**}**

**protected class GDestination implements Destination {**

**private String label;**

**private GDestination(String whereTo) {**

**label = whereTo;**

**}**

**public String readLabel() {**

**return label;**

**}**

**}**

**public Destination dest(String s) {**

**return new GDestination(s);**

**}**

**public Contents cont() {**

**return new Content();**

**}**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28**

**29**

**30**

**31**

**32**

**在这里我们给Goods类增加了一个private成员变量valueRate，意义是货物的价值系数，在内部类Content的方法value()计算价值时把它乘上。**

**我们发现，value()可以访问valueRate，这也是内部类的第二个好处 一个内部类对象可以访问创建它的外部类对象的内容，甚至包括私有变量！**

**这是一个非常有用的特性，为我们在设计时提供了更多的思路和捷径。要想实现这个功能，内部类对象就必须有指向外部类对象的引用。**

**内部类可以分为多种；主要以下4种:静态内部类，成员内部类，局部内部类，匿名内部类**

**静态内部类**

**静态内部类是指被声明为static的内部类，他可以不依赖内部类而实例，而通常的内部类需要实例化外部类，从而实例化。静态内部类不可以有与外部类有相同的类名。不能访问外部类的普通成员变量，但是可以访问静态成员变量和静态方法（包括私有类型）**

**成员内部类**

**一个 静态内部类去掉static 就是成员内部类，他可以自由的引用外部类的属性和方法，无论是静态还是非静态。但是不可以有静态属性和方法**

**局部内部类（定义在方法内的）**

**定义在一个代码块的内类,他的作用范围是所在代码块，是内部类中最少使用的一类型。局部内部类跟局部变量一样，不能被public ，protected，private以及static修饰，只能访问方法中定义final类型的局部变量**

**匿名内部类**

**匿名内部类是一种没有类名的内部类，不使用class，extends，implements，没有构造函数，他必须继承其他类或实现其他接口。匿名内部类的好处是使代码更加简洁，紧凑，但是带来的问题是易读性下降。**

**内部类的使用时机**

**1、实现事件监听器的时候（比方说actionListener 。。。采用内部类很容易实现）；**

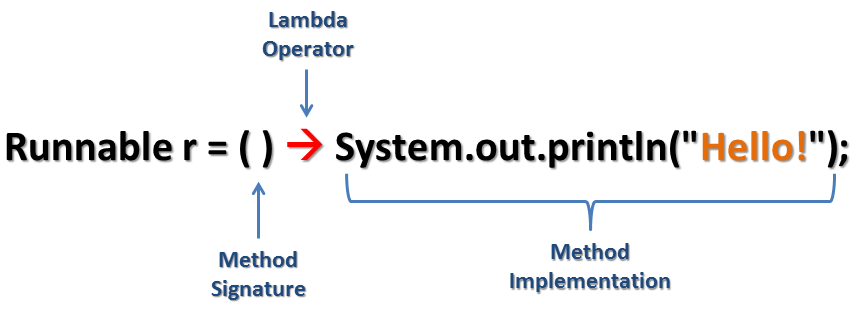
**2、编写事件驱动时（内部类的对象可以访问外部类的成员方法和变量，注意包括私有成员）；**

**3、在能实现功能的情况下，为了节省编译后产生的字节码（内部类可以减少字节码文件,即java文件编译后的.class文件）；**

# 47.多线程callable ,runnable，futrueTask ,thread

# 48.函数表达式lambda

从java8出现以来lambda是最重要的特性之一，它可以让我们用简洁流畅的代码完成一个功能。 很长一段时间java被吐槽是冗余和缺乏函数式编程能力的语言，随着函数式编程的流行java8种也引入了 这种编程风格。



## 什么是lambda?

lambda表达式是一段可以传递的代码，它的核心思想是将面向对象中的传递数据变成传递行为。 我们回顾一下在使用java8之前要做的事，之前我们编写一个线程时是这样的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Runnable r = new Runnable() {      @Override      public void run() {          System.out.println("do something.");      }  } |

也有人会写一个类去实现Runnable接口，这样做没有问题，我们注意这个接口中只有一个run方法， 当把Runnable对象给Thread对象作为构造参数时创建一个线程，运行后将输出do something.。 我们使用匿名内部类的方式实现了该方法。

这实际上是一个代码即数据的例子，在run方法中是线程要执行的一个任务，但上面的代码中任务内容已经被规定死了。 当我们有多个不同的任务时，需要重复编写如上代码。

设计匿名内部类的目的，就是为了方便 Java 程序员将代码作为数据传递。不过，匿名内部 类还是不够简便。 为了执行一个简单的任务逻辑，不得不加上 6 行冗繁的样板代码。那如果是lambda该怎么做?

|  |  |
| --- | --- |
| 1 | Runnable r = () -> System.out.println("do something."); |

嗯，这代码看起来很酷，你可以看到我们用()和->的方式完成了这件事，这是一个没有名字的函数，也没有人和参数，再简单不过了。 使用->将参数和实现逻辑分离，当运行这个线程的时候执行的是->之后的代码片段，且编译器帮助我们做了类型推导； 这个代码片段可以是用{}包含的一段逻辑。下面一起来学习一下lambda的语法。

## 基础语法

在lambda中我们遵循如下的表达式来编写：

|  |  |
| --- | --- |
| 1 | expression = (variable) -> action |

* variable: 这是一个变量,一个占位符。像x,y,z,可以是多个变量；
* action: 这里我称它为action, 这是我们实现的代码逻辑部分,它可以是一行代码也可以是一个代码片段。

可以看到Java中lambda表达式的格式：参数、箭头、以及动作实现，当一个动作实现无法用一行代码完成，可以编写 一段代码用{}包裹起来。

lambda表达式可以包含多个参数,例如：

|  |  |
| --- | --- |
| 1 | int sum = (x, y) -> x + y; |

这时候我们应该思考这段代码不是之前的x和y数字相加，而是创建了一个函数，用来计算两个操作数的和。 后面用int类型进行接收，在lambda中为我们省略去了return。

## 函数式接口

函数式接口是只有一个方法的接口，用作lambda表达式的类型。前面写的例子就是一个函数式接口，来看看jdk中的Runnable源码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @FunctionalInterface  public interface Runnable {      /\*\*       \* When an object implementing interface <code>Runnable</code> is used       \* to create a thread, starting the thread causes the object's       \* <code>run</code> method to be called in that separately executing       \* thread.       \* <p>       \* The general contract of the method <code>run</code> is that it may       \* take any action whatsoever.       \*       \* @see     java.lang.Thread#run()       \*/      public abstract void run();  } |

这里只有一个抽象方法run，实际上你不写public abstract也是可以的，在接口中定义的方法都是public abstract的。 同时也使用注解@FunctionalInterface告诉编译器这是一个函数式接口，当然你不这么写也可以，标识后明确了这个函数中 只有一个抽象方法，当你尝试在接口中编写多个方法的时候编译器将不允许这么干。

## 尝试函数式接口

我们来编写一个函数式接口，输入一个年龄，判断这个人是否是成人。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class FunctionInterfaceDemo {      @FunctionalInterface      interface Predicate<T> {          boolean test(T t);      }      /\*\*       \* 执行Predicate判断       \*       \* @param age       年龄       \* @param predicate Predicate函数式接口       \* @return          返回布尔类型结果       \*/      public static boolean doPredicate(int age, Predicate<Integer> predicate) {          return predicate.test(age);      }        public static void main(String[] args) {          boolean isAdult = doPredicate(20, x -> x >= 18);          System.out.println(isAdult);      }  } |

从这个例子我们很轻松的完成 是否是成人 的动作，其次判断是否是成人，在此之前我们的做法一般是编写一个 判断是否是成人的方法，是无法将 判断 共用的。而在本例只，你要做的是将 行为 (判断是否是成人，或者是判断是否大于30岁) 传递进去，函数式接口告诉你结果是什么。

实际上诸如上述例子中的接口，伟大的jdk设计者为我们准备了java.util.function包

我们前面写的Predicate函数式接口也是JDK种的一个实现，他们大致分为以下几类：

### 消费型接口示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public static void donation(Integer money, Consumer<Integer> consumer){      consumer.accept(money);  }  public static void main(String[] args) {      donation(1000, money -> System.out.println("好心的麦乐迪为Blade捐赠了"+money+"元")) ;  } |

### 供给型接口示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static List<Integer> supply(Integer num, Supplier<Integer> supplier){         List<Integer> resultList = new ArrayList<Integer>()   ;         for(int x=0;x<num;x++)             resultList.add(supplier.get());         return resultList ;  }  public static void main(String[] args) {      List<Integer> list = supply(10,() -> (int)(Math.random()\*100));      list.forEach(System.out::println);  } |

### 函数型接口示例

转换字符串为Integer

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public static Integer convert(String str, Function<String, Integer> function) {      return function.apply(str);  }  public static void main(String[] args) {      Integer value = convert("28", x -> Integer.parseInt(x));  } |

### 断言型接口示例

筛选出只有2个字的水果

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static List<String> filter(List<String> fruit, Predicate<String> predicate){      List<String> f = new ArrayList<>();      for (String s : fruit) {          if(predicate.test(s)){              f.add(s);          }      }      return f;  }  public static void main(String[] args) {      List<String> fruit = Arrays.asList("香蕉", "哈密瓜", "榴莲", "火龙果", "水蜜桃");      List<String> newFruit = filter(fruit, (f) -> f.length() == 2);      System.out.println(newFruit);  } |

## 默认方法

在Java语言中，一个接口中定义的方法必须由实现类提供实现。但是当接口中加入新的API时， 实现类按照约定也要修改实现，而Java8的API对现有接口也添加了很多方法，比如List接口中添加了sort方法。 如果按照之前的做法，那么所有的实现类都要实现sort方法，JDK的编写者们一定非常抓狂。

幸运的是我们使用了Java8，这一问题将得到很好的解决，在Java8种引入新的机制，支持在接口中声明方法同时提供实现。 这令人激动不已，你有两种方式完成 1.在接口内声明静态方法 2.指定一个默认方法。

我们来看看在JDK8中上述List接口添加方法的问题是如何解决的

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | default void sort(Comparator<? super E> c) {      Object[] a = this.toArray();      Arrays.sort(a, (Comparator) c);      ListIterator<E> i = this.listIterator();      for (Object e : a) {          i.next();          i.set((E) e);      }  } |

翻阅List接口的源码，其中加入一个默认方法default void sort(Comparator<? super E> c)。 在返回值之前加入default关键字，有了这个方法我们可以直接调用sort方法进行排序。

|  |  |
| --- | --- |
| 1  2  3 | List<Integer> list = Arrays.asList(2, 7, 3, 1, 8, 6, 4);  list.sort(Comparator.naturalOrder());  System.out.println(list); |

Comparator.naturalOrder()是一个自然排序的实现，这里可以自定义排序方案。你经常看到使用Java8操作集合的时候可以直接foreach的原因也是在Iterable接口中也新增了一个默认方法：forEach，该方法功能和 for 循环类似，但是允许 用户使用一个Lambda表达式作为循环体。

例子:

public class LambdaTest {  
  
  
 public static void main(String[] arg){  
  
 int num = Runtime.getRuntime().availableProcessors();  
 System.out.println("处理器数: " + num);  
  
 //1.用lambda表达式实现Runnable  
 // Java 8之前：  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("Before Java8, too much code for too little to do");  
 }  
 }).start();  
 //Java 8方式：  
 new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();  
 new Thread( () -> {  
 System.out.println("In Java8, Lambda expression rocks !!");  
 }).start();  
  
  
 //2.使用lambda表达式对列表进行迭代  
 String[] atp = {"Rafael Nadal", "Novak Djokovic",  
 "Stanislas Wawrinka",  
 "David Ferrer","Roger Federer",  
 "Andy Murray","Tomas Berdych",  
 "Juan Martin Del Potro"};  
 List<String> players = Arrays.asList(atp);  
 // 以前的循环方式  
 for (String player : players) {  
 System.out.print(player + "; ");  
 }  
  
 // 使用 lambda 表达式以及函数操作(functional operation)  
 players.forEach((player) -> System.out.print(player + "; "));  
 // 在 Java 8 中使用双冒号操作符(double colon operator)  
 players.forEach(System.out::println);  
  
  
 //3.使用Lambdas排序集合  
 //使用匿名内部类进行排序  
// players.sort(new Comparator<String>() {  
// @Override  
// public int compare(String o1, String o2) {  
// return (o1.compareTo(o2));  
// }  
// });  
 // System.out.println("排序后: ");  
 // players.forEach((player) -> System.out.print(player + "; "));  
 //使用Lambda  
 Comparator<String> sortByName = (String s1, String s2) -> (s1.compareTo(s2));  
 Collections.sort(players, sortByName);  
 System.out.println("排序后: ");  
 players.forEach((player) -> System.out.print(player + "; "));  
  
  
 //使用lambda表达式和函数式接口Predicate  
 System.out.println("测试Predicate");  
 testPredicate();  
  
  
  
 }  
  
  
 public static void testPredicate(){  
 List<Person> javaProgrammers = new ArrayList<Person>() {  
 {  
 add(new Person("Elsdon", "Jaycob", "Java programmer", "male", 43, 2000));  
 add(new Person("Tamsen", "Brittany", "Java programmer", "female", 23, 1500));  
 add(new Person("Floyd", "Donny", "Java programmer", "male", 33, 1800));  
 add(new Person("Sindy", "Jonie", "Java programmer", "female", 32, 1600));  
 add(new Person("Vere", "Hervey", "Java programmer", "male", 22, 1200));  
 add(new Person("Maude", "Jaimie", "Java programmer", "female", 27, 1900));  
 add(new Person("Shawn", "Randall", "Java programmer", "male", 30, 2300));  
 add(new Person("Jayden", "Corrina", "Java programmer", "female", 35, 1700));  
 add(new Person("Palmer", "Dene", "Java programmer", "male", 33, 2000));  
 add(new Person("Addison", "Pam", "Java programmer", "female", 34, 1300));  
 }  
 };  
  
 List<Person> phpProgrammers = new ArrayList<Person>() {  
 {  
 add(new Person("Jarrod", "Pace", "PHP programmer", "male", 34, 1550));  
 add(new Person("Clarette", "Cicely", "PHP programmer", "female", 23, 1200));  
 add(new Person("Victor", "Channing", "PHP programmer", "male", 32, 1600));  
 add(new Person("Tori", "Sheryl", "PHP programmer", "female", 21, 1000));  
 add(new Person("Osborne", "Shad", "PHP programmer", "male", 32, 1100));  
 add(new Person("Rosalind", "Layla", "PHP programmer", "female", 25, 1300));  
 add(new Person("Fraser", "Hewie", "PHP programmer", "male", 36, 1100));  
 add(new Person("Quinn", "Tamara", "PHP programmer", "female", 21, 1000));  
 add(new Person("Alvin", "Lance", "PHP programmer", "male", 38, 1600));  
 add(new Person("Evonne", "Shari", "PHP programmer", "female", 40, 1800));  
 }  
 };  
  
  
 System.out.println("所有java程序员的姓名:");  
 javaProgrammers.forEach((p) ->System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
 System.out.println("所有php程序员的姓名:");  
 phpProgrammers.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
  
 //Consumer是java8的函数式接口  
 System.out.println("给程序员加薪 5% :");  
 Consumer<Person> giveRaise = e -> e.setSalary(e.getSalary() / 100 \* 5 + e.getSalary());  
 javaProgrammers.forEach(giveRaise);  
 phpProgrammers.forEach(giveRaise);  
  
  
  
 System.out.println("下面是月薪超过 $1,400 的PHP程序员:");  
 phpProgrammers.stream()  
 .filter((p) -> (p.getSalary() > 1400))  
 .forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
  
 // 定义 filters  
 Predicate<Person> ageFilter = (p) -> (p.getAge() > 25);  
 Predicate<Person> salaryFilter = (p) -> (p.getSalary() > 1400);  
 Predicate<Person> genderFilter = (p) -> ("female".equals(p.getGender()));  
  
 System.out.println("下面是年龄大于 24岁且月薪在$1,400以上的女PHP程序员:");  
 phpProgrammers.stream()  
 .filter(ageFilter)  
 .filter(salaryFilter)  
 .filter(genderFilter)  
 .forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
 // 重用filters  
 System.out.println("年龄大于 24岁的女性 Java programmers:");  
 javaProgrammers.stream()  
 .filter(ageFilter)  
 .filter(genderFilter)  
 .forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
  
 System.out.println("最前面的3个 Java programmers:");  
 javaProgrammers.stream()  
 .limit(3)  
 .forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
  
 System.out.println("最前面的3个女性 Java programmers:");  
 javaProgrammers.stream()  
 .filter(genderFilter)  
 .limit(3)  
 .forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));  
  
  
 //排序  
 System.out.println("根据 name 排序,并显示前5个 Java programmers:");  
 List<Person> sortedJavaProgrammers = javaProgrammers  
 .stream()  
 .sorted((p, p2) -> (p.getFirstName().compareTo(p2.getFirstName())))  
 .limit(5)  
 .collect(Collectors.toList());  
  
 sortedJavaProgrammers.forEach((p) -> System.out.printf("%s %s; %n", p.getFirstName(), p.getLastName()));  
  
 System.out.println("根据 salary 排序 Java programmers:");  
 sortedJavaProgrammers = javaProgrammers  
 .stream()  
 .sorted( (p, p2) -> (p.getSalary() - p2.getSalary()) )  
 .collect(Collectors.toList());  
  
 sortedJavaProgrammers.forEach((p) -> System.out.printf("%s %s; %n", p.getFirstName(), p.getLastName()));  
  
  
 //min和max方法  
 System.out.println("工资最低的 Java programmer:");  
 Person pers = javaProgrammers  
 .stream()  
 .min((p1, p2) -> (p1.getSalary() - p2.getSalary()))  
 .get();  
  
 System.out.printf("Name: %s %s; Salary: $%,d.", pers.getFirstName(), pers.getLastName(), pers.getSalary());  
  
 System.out.println("工资最高的 Java programmer:");  
 Person person = javaProgrammers  
 .stream()  
 .max((p, p2) -> (p.getSalary() - p2.getSalary()))  
 .get();  
  
 System.out.printf("Name: %s %s; Salary: $%,d.", person.getFirstName(), person.getLastName(), person.getSalary());  
  
  
  
  
 System.out.println("将 PHP programmers 的 first name 拼接成字符串:");  
 String phpDevelopers = phpProgrammers  
 .stream()  
 .map(Person::getFirstName)  
 .collect(Collectors.joining(" ; ")); // 在进一步的操作中可以作为标记(token)  
 System.out.println(phpDevelopers);  
  
 System.out.println("将 Java programmers 的 first name 存放到 Set:");  
 Set<String> javaDevFirstName = javaProgrammers  
 .stream()  
 .map(Person::getFirstName)  
 .collect(Collectors.toSet());  
  
 System.out.println("将 Java programmers 的 first name 存放到 TreeSet:");  
 TreeSet<String> javaDevLastName = javaProgrammers  
 .stream()  
 .map(Person::getLastName)  
 .collect(Collectors.toCollection(TreeSet::new));  
  
  
 //Streams 还可以是并行的(parallel). Stream是对集合的包装  
 System.out.println("计算付给 Java programmers 的所有money:");  
 int totalSalary = javaProgrammers  
 .parallelStream()  
 .mapToInt(p -> p.getSalary())  
 .sum();  
  
 //使用summaryStatistics方法获得stream 中元素的各种汇总数据。  
 // 接下来,我们可以访问这些方法,比如getMax, getMin, getSum或getAverage:  
 //计算 count, min, max, sum, and average for numbers  
 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
 IntSummaryStatistics stats = numbers  
 .stream()  
 .mapToInt((x) -> x)  
 .summaryStatistics();  
  
 System.out.println("List中最大的数字 : " + stats.getMax());  
 System.out.println("List中最小的数字 : " + stats.getMin());  
 System.out.println("所有数字的总和 : " + stats.getSum());  
 System.out.println("所有数字的平均值 : " + stats.getAverage());  
 }  
}

public class Person {  
  
 private String firstName, lastName, job, gender;  
 private int salary, age;  
  
 public Person(String firstName, String lastName, String job,  
 String gender, int age, int salary) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.gender = gender;  
 this.age = age;  
 this.job = job;  
 this.salary = salary;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getJob() {  
 return job;  
 }  
  
 public void setJob(String job) {  
 this.job = job;  
 }  
  
 public String getGender() {  
 return gender;  
 }  
  
 public void setGender(String gender) {  
 this.gender = gender;  
 }  
  
 public int getSalary() {  
 return salary;  
 }  
  
 public void setSalary(int salary) {  
 this.salary = salary;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

# 49.native(Java Native Interface)关键字

Native修饰的方法空方法体，只是由java调用，不由java代码实现，是由底层c语言调用：

例子;

Thread类中线程的启动Start()方法中调用了start0（），start（）就是c语言实现该方法，也就是线程启动

public synchronized void start() {  
 */\*\*  
 \* This method is not invoked for the main method thread or "system"  
 \* group threads created/set up by the VM. Any new functionality added  
 \* to this method in the future may have to also be added to the VM.  
 \*  
 \* A zero status value corresponds to state "NEW".  
 \*/* if (threadStatus != 0)  
 throw new IllegalThreadStateException();  
  
 /\* Notify the group that this thread is about to be started  
 \* so that it can be added to the group's list of threads  
 \* and the group's unstarted count can be decremented. \*/  
 group.add(this);  
  
 boolean started = false;  
 try {  
 start0();  
 started = true;  
 } finally {  
 try {  
 if (!started) {  
 group.threadStartFailed(this);  
 }  
 } catch (Throwable ignore) {  
 /\* do nothing. If start0 threw a Throwable then  
 it will be passed up the call stack \*/  
 }  
 }  
}

native 方法

private native void start0();

而线程启动一般用start(),与直接调run（）方法的区别是

run()方法:在本线程内调用该Runnable对象的run()方法，可以重复多次调用；

        start()方法:启动一个线程，调用该Runnable对象的run()方法，不能多次启动一个线程

# 50数据类型

float：2^23 = 8388608，一共七位，这意味着最多能有7位[有效数字](https://www.baidu.com/s?wd=%E6%9C%89%E6%95%88%E6%95%B0%E5%AD%97&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，但绝对能保证的为6位，也即float的精度为6~7位[有效数字](https://www.baidu.com/s?wd=%E6%9C%89%E6%95%88%E6%95%B0%E5%AD%97&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)；

double：2^52 = 4503599627370496，一共16位，同理，double的精度为15~16位。