# 多线程

## 基本概念：程序、进程、线程

### 基本概念

1. 程序(program)是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。
2. 进程(process)是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。—— 生命周期

* 如：运行中的QQ，运行中的MP3播放器
* 程序是静态的，进程是动态的
* 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域

3）线程(thread)，进程可进一步细化为线程，是一个程序内部的一条执行路径。

* 若一个进程同一时间并行执行多个线程，就是支持多线程的
* 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开

销小

* 一个进程中的多个线程共享相同的内存单元/内存地址空间，它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。

### 单核CPU和多核CPU的理解

1）单核CPU，其实是一种假的多线程，因为在一个时间单元内，也只能执行一个线程的任务。例如：虽然有多车道，但是收费站只有一个工作人员在收费，只有收了费才能通过，那么CPU就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为CPU时间单元特别短，因此感觉不出来

 2）如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）

 3）一个Java应用程序java.exe，其实至少有三个线程：main()主线程，gc()垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程

### 并行与并发

1）并行：多个CPU同时执行多个任务。比如：多个人同时做不同的事

2）并发：一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多个人做同一件事。

### 4、使用多线程的优点

背景：以单核CPU为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程（切换执行线程需要消耗时间）来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1）提高应用程序的响应。对图形化界面更有意义，可增强用户体验。

2）提高计算机系统CPU的利用率

3）改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

### 5、何时需要多线程

1）程序需要同时执行两个或多个任务。

 2）程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。

 3）需要一些后台运行的程序时。

## 线程的创建和使用（重点）

### 线程的创建和启动

1）Java语言的JVM允许程序运行多个线程，它通过java.lang.Thread类来体现。 

2）Thread类的特性

* 每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为线程体
* 通过该Thread对象的start()方法来启动这个线程，而非直接调用run()

### API中创建线程的两种方式

#### 方式一：继承Thread类

1) 定义子类继承Thread类。

2) 子类中重写Thread类中的run方法。

3) 创建Thread子类对象，即创建了线程对象。

4) 调用线程对象start方法：启动线程，调用run方法。

例子：遍历100以内的所有的偶数

//1.创建一个继承于Thread类的子类

class MyThread extends Thread {

//2.重写Thread类的run()

@Override

public void run() {

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

if(i % 2 == 0){

System.out.println(Thread.currentThread().getName() + ":" + i);

}}}}

public class ThreadTest {

public static void main(String[] args) {

//3.创建Thread类的子类的对象

MyThread t1 = new MyThread();

//4.通过此对象调用start()：①启动当前线程； ②调用当前线程的run()

t1.start();

//问题一：不能通过直接调用run()的方式启动线程，此时只单纯调用run()方法，未启动线程

//t1.run();

//问题二：再启动一个线程，遍历100以内的偶数。不可以还让已经start()的线程去执行。会报IllegalThreadStateException

//t1.start();

//此时需要重新创建一个线程的对象

MyThread t2 = new MyThread();

t2.start();

//如下操作仍然是在main线程中执行的。

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

if(i % 2 == 0){

System.out.println(Thread.currentThread().getName() + ":" + i + "\*\*\*\*\*\*\*\*\*\*\*main()\*\*\*\*\*\*\*\*\*\*\*\*");

}}}}

练习：创建两个分线程，其中一个线程遍历100以内的偶数，另一个线程遍历100以内的奇数

public static void main(String[] args) {

// MyThread1 m1 = new MyThread1();

// MyThread2 m2 = new MyThread2();

// m1.start();

// m2.start();

//创建Thread类的匿名子类的方式

new Thread(){

@Override

public void run() {

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

if(i % 2 == 0){

System.out.println(Thread.currentThread().getName() + ":" + i);

}}}}.start();

new Thread(){

@Override

public void run() {

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

if(i % 2 != 0){

System.out.println(Thread.currentThread().getName() + ":" + i);

}}}}.start();

}

#### 方式二：实现Runnable接口

1) 创建一个实现了Runnable接口的类。

2) 实现类去实现Runnable中的抽象方法：run()。

3) 创建实现类的对象。

4) 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象。

5) 通过Thread类的对象调用start()。

//1.创建一个实现了Runnable接口的类

class MThread implements Runnable{

//2.实现类去实现Runnable中的抽象方法：run()

@Override

public void run() {

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

if(i % 2 == 0){

System.out.println(Thread.currentThread().getName() + ":" + i);

}}}}

public class ThreadTest1 {

public static void main(String[] args) {

//3.创建实现类的对象

MThread mThread = new MThread();

//4.将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象

Thread t1 = new Thread(mThread);

t1.setName("线程1");

//5.通过Thread类的对象调用start()：①启动线程；②调用当前线程（Thread）的run()-->调用了Runnable类型的target（mThread）的run()

t1.start();

//再启动一个线程，遍历100以内的偶数

Thread t2 = new Thread(mThread);

t2.setName("线程2");

t2.start();

}

}

#### 比较创建线程的两种方式

开发中：优先选择：实现Runnable接口的方式

原因：1. 实现的方式没有类的单继承性的局限性

2. 实现的方式更适合来处理多个线程有共享数据的情况。

联系：public class Thread implements Runnable

相同点：两种方式都需要重写run(),将线程要执行的逻辑声明在run()中。

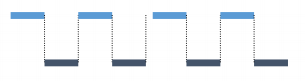
### Thread类的常用方法

1. start():启动当前线程；调用当前线程的run()
2. run(): 通常需要重写Thread类中的此方法，将创建的线程要执行的操作声明在此方法中
3. currentThread():静态方法，返回执行当前代码的线程
4. getName():获取当前线程的名字，默认为Thread-0，依次递增
5. setName():设置当前线程的名字，给主线程命名：Thread.currentThread().setName("主线程")
6. yield():释放当前cpu的执行权
7. join():在线程a中调用线程b的join(),此时线程a就进入阻塞状态，直到线程b完全执行完以后，线程a才结束阻塞状态。
8. stop():已过时。当执行此方法时，强制结束当前线程。
9. sleep(long millitime):让当前线程“睡眠”指定的millitime毫秒。在指定的millitime毫秒时间内，当前线程是阻塞状态。
10. isAlive():判断当前线程是否存活

### 线程的调度

1）调度策略：

* 时间片



* 抢占式：高优先级的线程抢占CPU

2）Java的调度方法：

* 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
* 对高优先级，使用优先调度的抢占式策略

### 线程的优先级

1. 线程的优先级等级

 MAX\_PRIORITY：10

 MIN \_PRIORITY：1

 NORM\_PRIORITY：5

1. 涉及的方法

 getPriority()：返回线程优先值

 setPriority(int newPriority)：改变线程的优先级，在start()方法前设置

1. 说明

* 线程创建时继承父线程的优先级
* 高优先级的线程要抢占低优先级线程cpu的执行权。但是只是从概率上讲，高优先级的线程高概率的情况下被执行。并不意味着只有当高优先级的线程执行完以后，低优先级的线程才执行

### 补充：线程的分类

Java中的线程分为两类：一种是守护线程，一种是用户线程

1）它们在几乎每个方面都是相同的，唯一的区别是判断JVM何时离开

 2）守护线程是用来服务用户线程的，通过在start()方法前调用thread.setDaemon(true)可以把一个用户线程变成一个守护线程

 3）Java垃圾回收就是一个典型的守护线程

 4）若JVM中都是守护线程，当前JVM将退出 

5）形象理解：兔死狗烹，鸟尽弓藏

## 线程的生命周期

**JDK中用Thread.State类定义了线程的几种状态**

要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用Thread类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态： 

1）新建： 当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态

 2）就绪：处于新建状态的线程被start()后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源

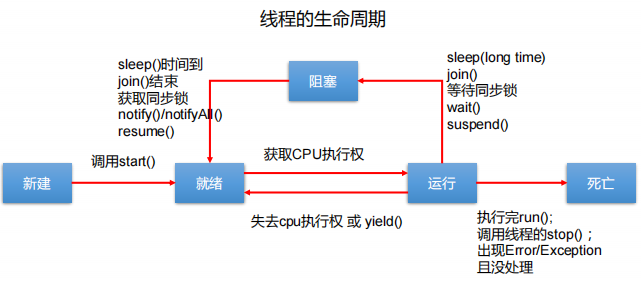
 3）运行：当就绪的线程被调度并获得CPU资源时,便进入运行状态， run()方法定义了线

程的操作和功能

 4）阻塞：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出 CPU 并临时中

止自己的执行，进入阻塞状态

 5）死亡：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束



## 线程的同步（重点）

### 1、多线程安全问题

1）问题的原因：

当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有

执行完，另一个线程参与进来执行。导致共享数据的错误。

2）解决办法：

对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以

参与执行。

### 同步机制解决线程安全问题

#### 方式一：同步代码块

synchronized(同步监视器){

//需要被同步的代码

}

说明：

1. 操作共享数据的代码，即为需要被同步的代码。-->不能包含代码多了，也不能包含代码少了

* 范围太小：没锁住所有有安全问题的代码
* 范围太大：没发挥多线程的功能

2）共享数据：多个线程共同操作的变量。比如：ticket就是共享数据。

3）同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。

要求：多个线程必须要共用同一把锁。

注意：

* 同步的方式，解决了线程的安全问题。---好处
* 操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低。 ---局限性

补充：在实现Runnable接口创建多线程的方式中，我们可以考虑使用this充当同步监视器

class Window1 implements Runnable{

private int ticket = 100;

// Object obj = new Object();

// Dog dog = new Dog();

@Override

public void run() {

//Object obj = new Object(); 此时obj不唯一，未解决线程安全问题

while(true){

synchronized (this){//此时的this:唯一的Window1的对象

//方式二：synchronized (dog) {

//方式三：synchronized (obj) {

if (ticket > 0) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + ":卖票，票号为：" + ticket);

ticket--;

} else {

break;

}}}}}

public class WindowTest1 {

public static void main(String[] args) {

Window1 w = new Window1();

Thread t1 = new Thread(w);

Thread t2 = new Thread(w);

Thread t3 = new Thread(w);

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

t2.start();

t3.start();

}

}

class Dog{}

使用同步代码块解决继承Thread类的方式的线程安全问题，此时慎用this充当同步监视器，考虑使用当前类充当同步监视器

class Window2 extends Thread{

private static int ticket = 100;

private static Object obj = new Object();（static保证obj唯一）

@Override

public void run() {

while(true){

//正确的方式

//synchronized (obj){

synchronized (Window2.class){

//Class clazz = Window2.class,Window2.class只会加载一次

//错误的方式：this代表着t1,t2,t3三个对象

//synchronized (this){

if(ticket > 0){

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(getName() + "：卖票，票号为：" + ticket);

ticket--;

}else{

break;

}}}}}

public class WindowTest2 {

public static void main(String[] args) {

Window2 t1 = new Window2();

Window2 t2 = new Window2();

Window2 t3 = new Window2();

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

t2.start();

t3.start();

}}

#### 方式二：同步方法

如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明同步的（synchronize）。

关于同步方法的总结：

1）同步方法仍然涉及到同步监视器，只是不需要我们显式的声明

2）非静态的同步方法，同步监视器是：this --- 适用于实现

3）静态的同步方法，同步监视器是：当前类本身 --- 适用于继承

使用同步方法解决实现Runnable接口的线程安全问题

class Window3 implements Runnable {

private int ticket = 100;

@Override

public void run() {

while (true) {

show();

}

}

private synchronized void show(){//同步监视器：this

//synchronized (this){此为同步代码块

if (ticket > 0) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + ":卖票，票号为：" + ticket);

ticket--;

}

//}

}}

public class WindowTest3 {

public static void main(String[] args) {

Window3 w = new Window3();

Thread t1 = new Thread(w);

Thread t2 = new Thread(w);

Thread t3 = new Thread(w);

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

t2.start();

t3.start();

}}

使用同步方法处理继承Thread类的方式中的线程安全问题

class Window4 extends Thread {

private static int ticket = 100;

@Override

public void run() {

while (true) {

show();

}

}

private **static** synchronized void show(){//同步监视器：Window4.class

//private synchronized void show(){//同步监视器（此时默认也为this）：t1,t2,t3。此种解决方式是错误的

if (ticket > 0) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + "：卖票，票号为：" + ticket);

ticket--;

}}}

public class WindowTest4 {

public static void main(String[] args) {

Window4 t1 = new Window4();

Window4 t2 = new Window4();

Window4 t3 = new Window4();

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

t2.start();

t3.start();

}}

### 释放锁的操作

1）当前线程的同步方法、同步代码块执行结束。

 2）当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。

 3）当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束。

 4）当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。

### 不会释放锁的操作

1）线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行

2）线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）

 应尽量避免使用suspend()和resume()来控制线程

### 线程的死锁问题

1. 死锁：

* 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
* 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续

 2）解决方法：

* 专门的算法、原则
* 尽量减少同步资源的定义
* 尽量避免嵌套同步

StringBuffer s1 = new StringBuffer();

StringBuffer s2 = new StringBuffer();

new Thread(){

@Override

public void run() {

synchronized (s1){

s1.append("a");

s2.append("1");

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

synchronized (s2){

s1.append("b");

s2.append("2");

System.out.println(s1);

System.out.println(s2);

}}}}.start();

new Thread(new Runnable() {

@Override

public void run() {

synchronized (s2){

s1.append("c");

s2.append("3");

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

synchronized (s1){

s1.append("d");

s2.append("4");

System.out.println(s1);

System.out.println(s2);

}}}}).start();

说明：1、线程1在等待线程2释放锁s2，线程2在等待线程1释放锁s1，导致存在死锁；

2、若均未加等待时间，执行结果可能是abcd、1234，或者cdab、3412

### Lock锁解决线程安全问题

从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同

步锁对象来实现同步。同步锁使用Lock对象充当

java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象

ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁

class Window implements Runnable{

private int ticket = 100;

//1.实例化ReentrantLock

private ReentrantLock lock = new ReentrantLock();

@Override

public void run() {

while(true){

try{

//2.调用锁定方法lock()

lock.lock();

if(ticket > 0){

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + "：售票，票号为：" + ticket);

ticket--;

}else{

break;

}

}finally {

//3.调用解锁方法：unlock()

lock.unlock();

}}}}

public class LockTest {

public static void main(String[] args) {

Window w = new Window();

Thread t1 = new Thread(w);

Thread t2 = new Thread(w);

Thread t3 = new Thread(w);

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

t2.start();

t3.start();

}}

### synchronized与Lock的异同

1）相同点：二者都可以解决线程安全问题

2）不同点：synchronized机制在执行完相应的同步代码以后，自动的释放同步监视器

Lock需要手动的启动同步（lock()），同时结束同步也需要手动的实现（unlock()）

3）优先使用顺序：

Lock --- 同步代码块（已经进入了方法体，分配了相应资源）--- 同步方法（在方法体之外）

### 练习题

有两个储户分别向同一个账户存3000元，每次存1000，存3次。每次存完打印账户余额。

class Account{

private double balance;

public Account(double balance) {

this.balance = balance;

}

//存钱

public **synchronized** void deposit(double amt){

if(amt > 0){

balance += amt;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + ":存钱成功。余额为：" + balance);

}}}

class Customer **extends Thread**{

private Account acct;

public Customer(Account acct) {

this.acct = acct;

}

@Override

public void run() {

for (int i = 0; i < 3; i++) {

**acct.deposit(1000);**

}}}

public class AccountTest {

public static void main(String[] args) {

Account acct = new Account(0);

Customer c1 = new Customer(acct);

Customer c2 = new Customer(acct);

c1.setName("甲");

c2.setName("乙");

c1.start();

c2.start();

}}

## 线程的通信

### wait() 方法

1）在当前线程中调用方法： 对象名.wait()

2）使当前线程进入等待（某对象）状态 ，直到另一线程对该对象发出 notify (或notifyAll) 为止

3）调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

4）调用此方法后，当前线程将释放对象监控权 ，然后进入等待

5）在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行

### notify()/notifyAll()方法

1）在当前线程中调用方法： 对象名.notify()

2）功能：唤醒等待该对象监控权的一个/所有线程 

3）调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

### 3、总结这三个方法

1）wait():一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。

2）notify():一旦执行此方法，就会唤醒被wait的一个线程。如果有多个线程被wait，就唤醒优先级高的那个。

3）notifyAll():一旦执行此方法，就会唤醒所有被wait的线程。

4）wait()，notify()，notifyAll()三个方法必须使用在同步代码块或同步方法中。

5）wait()，notify()，notifyAll()三个方法的调用者必须是同步代码块或同步方法中的同步监视器。否则，会出现IllegalMonitorStateException异常

6）wait()，notify()，notifyAll()三个方法是定义在java.lang.Object类中。

举例：使用两个线程打印 1-100。线程1, 线程2 交替打印

class Number implements Runnable{

private int number = 1;

private Object obj = new Object();

@Override

public void run() {

while(true){

synchronized (obj) {

obj.notify(); //唤醒被wait的一个线程

if(number <= 100){

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+":"+number);

number++;

try {

//使得调用如下wait()方法的线程进入阻塞状态，并释放同步监视器

obj.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}else{

break;

}}}}}

public class CommunicationTest {

public static void main(String[] args) {

Number number = new Number();

Thread t1 = new Thread(number);

Thread t2 = new Thread(number);

t1.setName("线程1");

t2.setName("线程2");

t1.start();

t2.start();

}}

### 4、面试题：sleep() 和 wait()的异同

1）相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。

2）不同点：

* 两个方法声明的位置不同：Thread类中声明sleep() , Object类中声明wait()；
* 调用的要求不同：sleep()可以在任何需要的场景下调用。 wait()必须使用在同步代码块或同步方法中；
* 关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。

### 5、线程通信的应用：生产者/消费者问题

## JDK5.0新增线程创建方式

### 新增方式一：实现Callable接口

1、与使用Runnable相比， Callable功能更强大些

* 相比run()方法，可以有返回值
* 方法可以抛出异常
* 支持泛型的返回值
* 需要借助FutureTask类，比如获取返回结果（使用get()方法）

2、Future接口

* 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等
* FutrueTask是Futrue接口的唯一的实现类
* FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

//1.创建一个实现Callable的实现类

class NumThread implements Callable{

//2.实现call方法，将此线程需要执行的操作声明在call()中

@Override

public Object call() throws Exception {

int sum = 0;

for (int i = 1; i <= 100; i++) {

if(i % 2 == 0){

System.out.println(i);

sum += i;

}

}

return sum;

}}

public class ThreadNew {

public static void main(String[] args) {

//3.创建Callable接口实现类的对象

NumThread numThread = new NumThread();

//4.将此Callable接口实现类的对象作为参数传递到FutureTask构造器中，创建FutureTask的对象

FutureTask futureTask = new FutureTask(numThread);

//5.将FutureTask的对象作为参数传递到Thread类的构造器中，创建Thread对象，并调用start()

new Thread(futureTask).start();

try {

//6.获取Callable中call方法的返回值

//get()返回值即为FutureTask构造器参数Callable实现类重写的call()的返回值

Object sum = futureTask.get();

System.out.println("总和为：" + sum);

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}}}

### 新增方式二：使用线程池

1、背景：经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大 

2、思路：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具

3、好处：

 1）提高响应速度（减少了创建新线程的时间）

 2）降低资源消耗（重复利用线程池中线程，不需要每次都创建）

 3）便于线程管理

* corePoolSize：核心池的大小
* maximumPoolSize：最大线程数 
* keepAliveTime：线程没有任务时最多保持多长时间后会终止

4、线程池相关API

1）JDK 5.0起提供了线程池相关API：ExecutorService 和 Executors

 2）ExecutorService：真正的线程池接口。常见子类ThreadPoolExecutor

* void execute(Runnable command)：执行任务/命令，没有返回值，一般用来执行Runnable
* <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable
* void shutdown()：关闭连接池

 3）Executors：工具类、线程池的工厂类，用于创建并返回不同类型的线程池

* Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
* Executors.newFixedThreadPool(n)：创建一个可重用固定线程数的线程池
* Executors.newSingleThreadExecutor()：创建一个只有一个线程的线程池
* Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行

class NumberThread implements Runnable{ //线程1

@Override

public void run() {

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

if(i % 2 == 0){

System.out.println(Thread.currentThread().getName() + ": " + i);

}}}}

class NumberThread1 implements Runnable{ //线程2

@Override

public void run() {

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

if(i % 2 != 0){

System.out.println(Thread.currentThread().getName() + ": " + i);

}}}}

public class ThreadPool {

public static void main(String[] args) {

//1.提供指定线程数量的线程池

ExecutorService service = Executors.newFixedThreadPool(10);

ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;

//设置线程池的属性

// System.out.println(service.getClass());

// service1.setCorePoolSize(15);

// service1.setKeepAliveTime();

//2.执行指定的线程的操作。需要提供实现Runnable接口或Callable接口实现类的对象

service.execute(new NumberThread());//适合适用于Runnable

service.execute(new NumberThread1());//适合适用于Runnable

// service.submit(Callable callable);//适合使用于Callable

//3.关闭连接池

service.shutdown();

}}

# 第11章 Java集合

## Java集合框架概述

### 集合概述

一方面， 面向对象语言对事物的体现都是以对象的形式，为了方便对多个对象的操作，就要对对象进行存储。另一方面，使用Array存储对象方面具有一些弊端，而Java集合就像一种容器，可以动态地把多个对象的引用放入容器中。

 Java 集合类可以用于存储数量不等的多个对象，还可用于保存具有映射关系的关联数组。

1）集合、数组都是对多个数据进行存储操作的结构，简称Java容器。

说明：此时的存储主要指的是内存层面的存储，不涉及到持久化的存储（.txt,.jpg,.avi，数据库中）

2）数组在存储多个数据方面的特点：

* 一旦初始化以后，其长度就确定了。
* 数组一旦定义好，其元素的类型也就确定了。我们也就只能操作指定类型的数据了。

比如：String[] arr;int[] arr1;Object[] arr2;

3）数组在存储多个数据方面的缺点：

* 一旦初始化以后，其长度就不可修改，不便于扩展。
* 数组中提供的方法非常有限，对于添加、删除、插入数据等操作，非常不便，同时效率不高。
* 获取数组中实际元素的个数的需求，数组没有现成的属性或方法可用
* 数组存储数据的特点：有序、可重复。对于无序、不可重复的需求，不能满足。

4）Java集合可分为Collection和Map两种体系

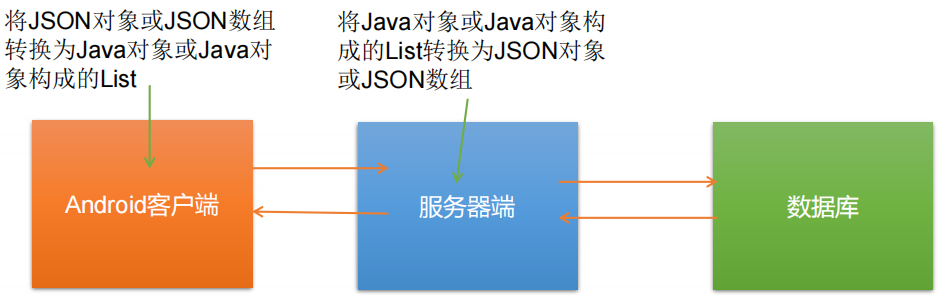
* Collection接口：单列集合，定义了存取一组对象的方法的集合

List：元素有序、可重复的集合 --- “动态”数组

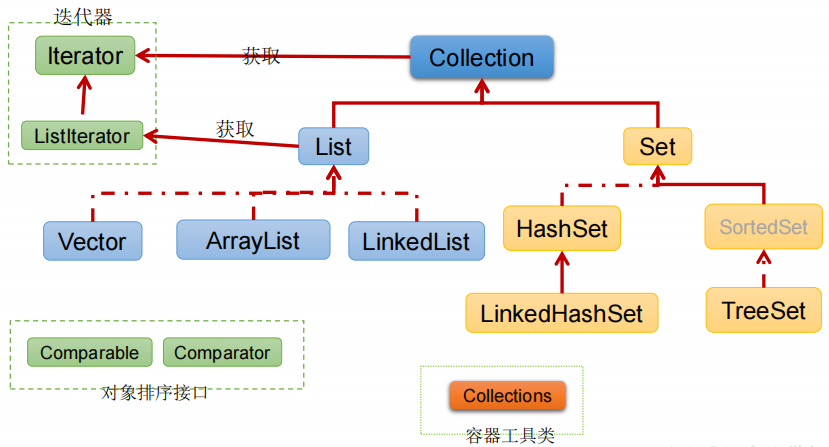
Set：元素无序、不可重复的集合 --- 高中讲的“集合”

* Map接口：双列集合，保存具有映射关系“key-value对”的集合 --- 高中函数：y = f(x)

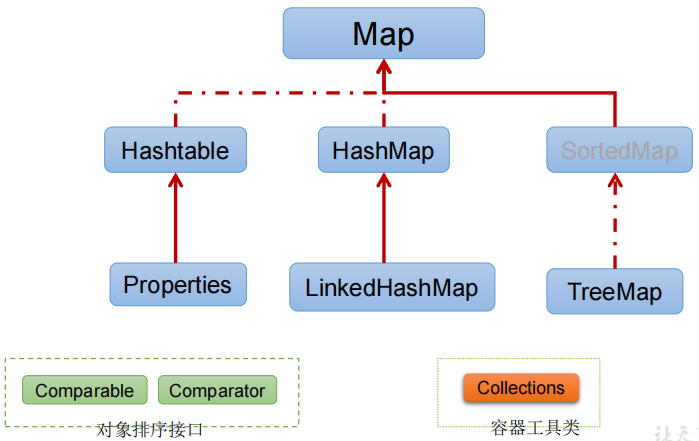
### 集合的使用场景



### Collection接口继承树



### Map接口继承树



## Collection接口方法

### 常用方法1：add、size、clear、isEmpty

Collection coll = **new** ArrayList();  
  
*//1、add(Object e):将元素e添加到集合coll中*coll.add(**"AA"**);  
coll.add(**"BB"**);  
coll.add(123); *//自动装箱*coll.add(**new** Date());  
*//2、size():获取添加的元素的个数*System.***out***.println(coll.size()); *//4  
//3、addAll(Collection coll1):将coll1集合中的元素添加到当前的集合中*Collection coll1 = **new** ArrayList();  
coll1.add(456);  
coll1.add(**"CC"**);  
coll.addAll(coll1);  
  
System.***out***.println(coll.size()); *//6*System.***out***.println(coll); *//[AA, BB, 123, Sat Jul 31 09:51:56 CST 2021, 456, CC]  
//4、clear():清空集合元素*coll.clear();  
*//5、isEmpty():判断当前集合是否为空*System.***out***.println(coll.isEmpty()); *//true*

### 常用方法2：contains、containsAll

向Collection接口的实现类的对象中添加数据obj时，要求obj所在类要重写equals()

Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
Person p = **new** Person(**"kid"**,30);  
coll.add(p);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
*//1.contains(Object obj):判断当前集合中是否包含obj  
//我们在判断时会调用obj对象所在类的equals()***boolean** contains = coll.contains(123);  
System.***out***.println(contains); *//true*System.***out***.println(coll.contains(**new** String(**"Tom"**))); *//true*System.***out***.println(coll.contains(p)); *//true*System.***out***.println(coll.contains(**new** Person(**"kid"**,30))); *//false -->true （重写equals方法，输出true）  
  
//2.containsAll(Collection coll1):判断形参coll1中的所有元素是否都存在于当前集合中*Collection coll1 = Arrays.*asList*(123,**"Tom"**);  
System.***out***.println(coll.containsAll(coll1)); *//true*

Collection coll1 = Arrays.*asList*(123,4567);  
System.***out***.println(coll.containsAll(coll1)); *//false*

### 常用方法3：remove、removeAll、retainAll、equals

*//1.remove(Object obj):从当前集合中移除obj元素。*Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
coll.remove(123);  
System.***out***.println(coll); *//[456, Person{name='Jerry', age=20}, Tom, false]*  
  
coll.remove(**new** Person(**"Jerry"**,20));  
System.***out***.println(coll); *//[456, Tom, false]*  
  
*//2. removeAll(Collection coll1):差集：从当前集合中移除coll1中所有的元素。*Collection coll1 = Arrays.*asList*(123,456);  
coll.removeAll(coll1);  
System.***out***.println(coll); *//[Tom, false]*

Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
*//3.retainAll(Collection coll1):交集：获取当前集合和coll1集合的交集，并返回给当前集合  
//*Collection coll1 = Arrays.*asList*(123,456,789);  
*//*coll.retainAll(coll1);  
*//*System.***out***.println(coll); *//[123, 456]  
  
//4.equals(Object obj):要想返回true，需要当前集合和形参集合的元素都相同。*Collection coll2 = **new** ArrayList();  
coll2.add(456);  
coll2.add(123);  
coll2.add(**new** Person(**"Jerry"**,20));  
coll2.add(**new** String(**"Tom"**));  
coll2.add(**false**);  
  
System.***out***.println(coll.equals(coll2)); *//false（ArrayList有序导致，若为hashSet则为true）*

### 常用方法4：toArray、asList

Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
*//1.集合 --->数组：toArray()*Object[] arr = coll.toArray();  
**for**(**int** i = 0;i < arr.**length**;i++){  
System.out.println(arr[i]);}  
  
*//2、拓展：数组 --->集合:调用Arrays类的静态方法asList()*List<String> list = Arrays.*asList*(**new** String[]{**"AA"**, **"BB"**, **"CC"**});  
System.***out***.println(list); *//[AA, BB, CC]*List arr1 = Arrays.*asList*(**new int**[]{123, 456});  
System.***out***.println(arr1.size()); *//1*List arr2 = Arrays.*asList*(**new** Integer[]{123, 456});  
System.***out***.println(arr2.size()); *//2*List arr3 = Arrays.*asList*(123, 456);  
System.***out***.println(arr3.size()); *//2*

## Iterator迭代器接口

### Iterator概述

1）Iterator对象称为迭代器(设计模式的一种)，主要用于遍历 Collection 集合中的元素。

2）GOF给迭代器模式的定义为：提供一种方法访问一个容器(container)对象中各个元素，而又不需暴露该对象的内部细节。迭代器模式，就是为容器而生。 

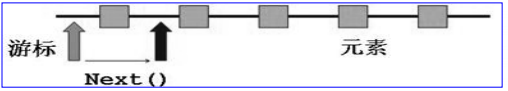
3）Collection接口继承java.lang.Iterable接口，该接口有一个iterator()方法，那么所有实现了Collection接口的集合类都有一个iterator()方法，用以返回一个实现了Iterator接口的对象。

4）Iterator 仅用于遍历集合，Iterator 本身并不提供承装对象的能力。如果需要创建Iterator 对象，则必须有一个被迭代的集合。

5）集合对象每次调用iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前。

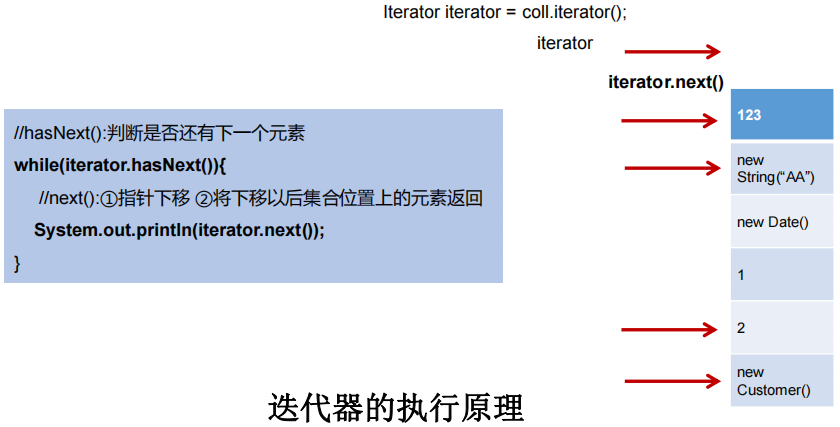
### Iterator的使用

在调用it.next()方法之前必须要调用it.hasNext()进行检测。若不调用，且下一条记录无效，直接调用it.next()会抛出NoSuchElementException异常



Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
Iterator iterator = coll.iterator();  
*//方式一：*System.***out***.println(iterator.next());  
System.***out***.println(iterator.next());  
System.***out***.println(iterator.next());  
System.***out***.println(iterator.next());  
System.***out***.println(iterator.next());  
*//报异常：NoSuchElementException*System.***out***.println(iterator.next());  
  
*//方式二：不推荐***for**(**int** i = 0;i < coll.size();i++){  
 System.***out***.println(iterator.next());  
}  
*//方式三：推荐  
//hasNext():判断是否还有下一个元素***while**(iterator.hasNext()){  
 *//next():①指针下移 ②将下移以后集合位置上的元素返回* System.***out***.println(iterator.next());  
}

### Iterator的执行原理



### 遍历集合的两种错误写法

Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
*//错误方式一：*Iterator iterator = coll.iterator();  
**while**((iterator.next()) != **null**){  
 System.***out***.println(iterator.next()); *//456、Tom（跳着输出），同时报错NoSuchElementException*}  
*//错误方式二：  
集合对象每次调用iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前***while** (coll.iterator().hasNext()){  
 System.***out***.println(coll.iterator().next()); *//无限输出第一个元素123*}

### remove()方法

1）Iterator可以删除集合的元素，但是是遍历过程中通过迭代器对象的remove方 法，不是集合对的remove方法。 

2）如果还未调用next()或在上一次调用 next 方法之后已经调用了 remove 方法，再调用remove都会报IllegalStateException。

Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
*//删除集合中"Tom"*Iterator iterator = coll.iterator();  
**while** (iterator.hasNext()){  
*// iterator.remove(); //报错IllegalStateException* Object obj = iterator.next();  
 **if**(**"Tom"**.equals(obj)){  
 iterator.remove();  
*// iterator.remove(); //报错IllegalStateException* }  
}  
*//遍历集合*iterator = coll.iterator(); *//回到起点*  
**while** (iterator.hasNext()){  
 System.***out***.println(iterator.next());  
}

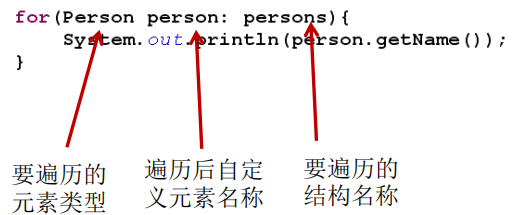
### foreach循环（增强for循环）

1）Java 5.0 提供了 foreach 循环迭代访问 Collection和数组。

2）遍历操作不需获取Collection或数组的长度，无需使用索引访问元素。

3）遍历集合的底层调用Iterator完成操作。

4）foreach还可以用来遍历数组。



Collection coll = **new** ArrayList();  
coll.add(123);  
coll.add(456);  
coll.add(**new** Person(**"Jerry"**,20));  
coll.add(**new** String(**"Tom"**));  
coll.add(**false**);  
  
*//for(集合元素的类型 局部变量 : 集合对象)***for**(Object obj : coll){  
 System.***out***.println(obj);  
}

**int**[] arr = **new int**[]{1,2,3,4,5,6};  
*//for(数组元素的类型 局部变量 : 数组对象)***for**(**int** i : arr){  
 System.***out***.println(i);  
}

1. 练习题

String[] arr = **new** String[]{**"MM"**,**"MM"**,**"MM"**};  
*//方式一：普通for赋值***for**(**int** i = 0;i < arr.**length**;i++){  
 arr[i] = **"GG"**;  
}  
**for**(**int** i = 0;i < arr.**length**;i++){  
 System.***out***.println(arr[i]); *//GG、GG、GG*}  
  
String[] arr1 = **new** String[]{**"MM"**,**"MM"**,**"MM"**};  
*//方式二：增强for循环***for**(String s : arr1){  
 s = **"DD"**;  
}  
**for**(**int** i = 0;i < arr1.**length**;i++){  
 System.***out***.println(arr1[i]); *//MM、MM、MM*}

## Collection子接口一：List

### List接口概述

1）鉴于Java中数组用来存储数据的局限性，我们通常使用List替代数组

2）List集合类中元素有序、且可重复，集合中的每个元素都有其对应的顺序索引

3）List容器中的元素都对应一个整数型的序号记载其在容器中的位置，可以根据序号存取容器中元素

JDK API中List接口的实现类常用的有：ArrayList、LinkedList和Vector

相同点：三个类都是实现了List接口，存储数据的特点相同：存储有序的、可重复的数据

不同点：

* ArrayList：作为List接口的主要实现类；线程不安全的，效率高；底层使用Object[] elementData存储
* LinkedList：对于频繁的插入、删除操作，使用此类效率比ArrayList高；底层使用双向链表存储
* Vector：作为List接口的古老实现类；线程安全的，效率低；底层使用Object[] elementData存储

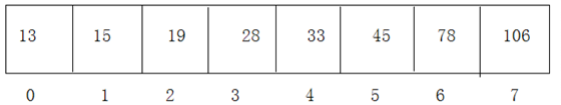
### List实现类之一：ArrayList

1. ArrayList 是 List 接口的典型实现类、主要实现类
2. 本质上，ArrayList是对象引用的一个”变长”数组
3. ArrayList的JDK1.8之前与之后的实现区别？

* JDK1.7：ArrayList像饿汉式，直接创建一个初始容量为10的数组
* JDK1.8：ArrayList像懒汉式，一开始创建一个长度为0的数组，当添加第一个元素时再创建一个始容量为10的数组

4）建议开发中使用带参的构造器：ArrayList list = new ArrayList(int capacity)

5）Arrays.asList(…)方法返回的List集合，既不是ArrayList实例，也不是Vector实例。 Arrays.asList(…)返回值是一个固定长度的List集合



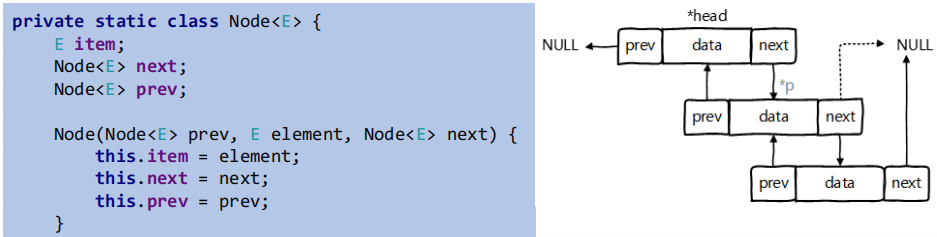
### List实现类之二：LinkedList

1）对于频繁的插入或删除元素的操作，建议使用LinkedList类，效率较高

2）LinkedList：双向链表，内部没有声明数组，而是定义了Node类型的first和last，用于记录首末元素。同时，定义内部类Node，作为LinkedList中保存数据的基本结构。

3）Node除了保存数据，还定义了两个变量：

* prev变量记录前一个元素的位置
* next变量记录下一个元素的位置



### List实现类之三：Vector

Vector是一个古老的集合，JDK1.0就有了。大多数操作与ArrayList相同，区别之处在于Vector是线程安全的。

在各种list中，最好把ArrayList作为缺省选择。当插入、删除频繁时，使用LinkedList；Vector总是比ArrayList慢，所以尽量避免使用。

### List接口方法

List除了从Collection集合继承的方法外，List 集合里添加了一些根据索引来操作集合元素的方法。

1. void add(int index, Object ele):在index位置插入ele元素
2. boolean addAll(int index, Collection eles):从index位置开始将eles中的所有元素添加进来
3. Object get(int index):获取指定index位置的元素

ArrayList list = **new** ArrayList();  
list.add(123);  
list.add(456);  
list.add(**"AA"**);  
list.add(**new** Person(**"Tom"**,12));  
list.add(456);  
  
*//void add(int index, Object ele):在index位置插入ele元素*list.add(1,**"BB"**); *//[123, BB, 456, AA, Person{name='Tom', age=12}, 456]*System.***out***.println(list);  
  
*boolean addAll(int index, Collection eles):从index位置开始将eles中的所有元素添加进来*List list1 = Arrays.*asList*(1, 2, 3);  
list.addAll(list1);  
*//list.add(list1); //把list1当做一个整体*System.***out***.println(list.size());*//9  
  
//Object get(int index):获取指定index位置的元素*System.***out***.println(list.get(0));

1. int indexOf(Object obj):返回obj在集合中首次出现的位置
2. int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置
3. Object remove(int index):移除指定index位置的元素，并返回此元素
4. Object set(int index, Object ele):设置指定index位置的元素为ele
5. List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置的子集合

ArrayList list = **new** ArrayList();  
list.add(123);  
list.add(456);  
list.add(**"AA"**);  
list.add(**new** Person(**"Tom"**,12));  
list.add(456);  
*//int indexOf(Object obj):返回obj在集合中首次出现的位置。如果不存在，返回-1.***int** index = list.indexOf(4567);  
System.***out***.println(index);  
  
*//int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置。如果不存在，返回-1.*System.***out***.println(list.lastIndexOf(456)); *//4  
  
//Object remove(int index):移除指定index位置的元素，并返回此元素*Object obj = list.remove(0);  
System.***out***.println(obj); *//123*System.***out***.println(list); *//[456, AA, Person{name='Tom', age=12}, 456]  
  
//Object set(int index, Object ele):设置指定index位置的元素为ele*list.set(1,**"CC"**);  
System.***out***.println(list); *//[456, CC, Person{name='Tom', age=12}, 456]  
  
//List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置的左闭右开区间的子集合*List subList = list.subList(2, 4);  
System.***out***.println(subList); *//[Person{name='Tom', age=12}, 456]*System.***out***.println(list); *//[456, CC, Person{name='Tom', age=12}, 456]*

总结：常用方法  
增：add(Object obj)  
删：remove(int index) / remove(Object obj)  
改：set(int index, Object ele)  
查：get(int index)  
插：add(int index, Object ele)  
长度：size()  
遍历：① Iterator迭代器方式  
 ② 增强for循环  
 ③ 普通的循环

### 面试题

1. remove(int index)和remove(Object obj)的区别

**public void** testListRemove() {  
 List list = **new** ArrayList();  
 list.add(1);  
 list.add(2);  
 list.add(3);  
 updateList(list);  
 System.***out***.println(list);  
}  
**private void** updateList(List list) {  
 list.remove(2); *//删除索引为2的数据，结果为[1,2]* list.remove(**new** Integer(2)); *//删除数据为2的值，结果为[1,3]*}

1. ArrayList/LinkedList/Vector的异同
2. ArrayList和LinkedList的异同

二者都线程不安全，相对线程安全的Vector，执行效率高。

此外，ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。对于

随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。对于新增

和删除操作add(特指插入)和remove，LinkedList比较占优势，因为ArrayList要移动数据。

1. ArrayList和Vector的区别

Vector和ArrayList几乎是完全相同的,唯一的区别在于Vector是同步类(synchronized)，属于

强同步类。因此开销就比ArrayList要大，访问要慢。正常情况下,大多数的Java程序员使用ArrayList而不是Vector,因为同步完全可以由程序员自己来控制。Vector每次扩容请求其大小的2倍空间，而ArrayList是1.5倍。Vector还有一个子类Stack。

## Collection子接口二：Set

### Set接口概述

1）Set接口是Collection的子接口，set接口没有提供额外的方法，使用的都是Collection中声明过的方法

2）Set集合不允许包含相同的元素，如果试把两个相同的元素加入同一个Set 集合中，则添加操作失败

3）Set判断两个对象是否相同不是使用 == 运算符，而是根据equals()方法

4）分类：

* HashSet：作为Set接口的主要实现类；线程不安全的；可以存储null值
* LinkedHashSet：作为HashSet的子类；遍历其内部数据时，可以按照添加的顺序遍历。对于频繁的遍历操作，LinkedHashSet效率高于HashSet
* TreeSet：可以按照添加对象的指定属性，进行排序

### Set实现类之一：HashSet

1）Set的无序性和不可重复性

* HashSet底层：数组+链表的结构
* 无序性：不等于随机性。存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值决定的。（每次遍历的结果顺序相同）
* 不可重复性：保证添加的元素按照equals()判断时，不能返回true（需重写equals方法、hashCode方法），即：相同的元素只能添加一个
* HashSet集合判断两个元素相等的标准：两个对象通过hashCode()方法比较相等，并且两个对象的 equals()方法返回值也相等。
* 对于存放在Set容器中的对象，对应的类一定要重写equals()和hashCode(Object obj)方法，以实现对象相等规则。即：“相等的对象必须具有相等的散列码”

Set set = **new** HashSet();  
set.add(456);  
set.add(123);  
set.add(123);  
set.add(**"AA"**);  
set.add(**"CC"**);  
set.add(**new** User(**"Tom"**,12));  
set.add(**new** User(**"Tom"**,12)); //需要重写equals方法、hashCode方法才会当做相同元素  
set.add(129);  
  
Iterator iterator = set.iterator();  
**while**(iterator.hasNext()){  
 System.***out***.println(iterator.next());  
}

输出：AA、CC、129、456、123、User{name='Tom', age=12}

2）向HashSet中添加元素的过程

当向HashSet集合中存入一个元素时，HashSet会调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据hashCode值，通过某种散列函数决定该对象在 HashSet 底层数组中的存储位置。（这个散列函数会与底层数组的长度相计算得到在数组中的下标，并且这种散列函数计算还尽可能保证能均匀存储元素，越是散列分布，该散列函数设计的越好） 

如果两个元素的hashCode()值相等，会再继续调用equals方法，如果equals方法结果为true，添加失败；如果为false，那么会保存该元素，但是该数组的位置已经有元素了，那么会通过链表的方式继续链接

如果两个元素的equals()方法返回true，但它们的hashCode()返回值不相等，hashSet将会把它们存储在不同的位置，但依然可以添加成功

我们向HashSet中添加元素a,首先调用元素a所在类的hashCode()方法，计算元素a的哈希值，

此哈希值接着通过某种算法计算出在HashSet底层数组中的存放位置（即为：索引位置），

判断数组此位置上是否已经有元素：

如果此位置上没有其他元素，则元素a添加成功。 ---> 情况1

如果此位置上有其他元素b(或以链表形式存在的多个元素），则比较元素a与元素b的hash值：

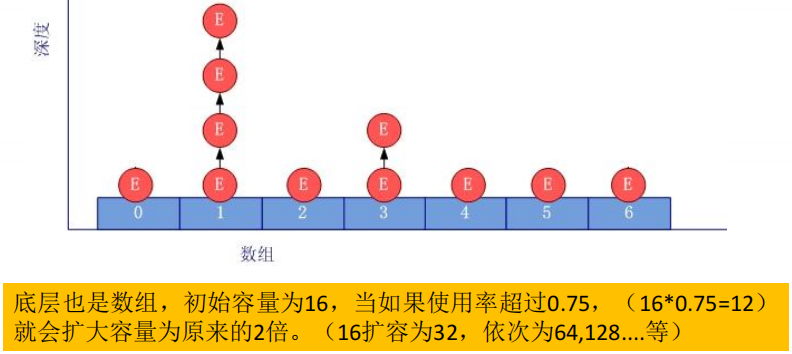
如果hash值不相同，则元素a添加成功。 ---> 情况2

如果hash值相同，进而需要调用元素a所在类的equals()方法：

equals()返回true,元素a添加失败

equals()返回false,则元素a添加成功。 ---> 情况3

对于添加成功的情况2和情况3而言：元素a与已经存在指定索引位置上数据以链表的方式存储



1. 重写hashCode()方法的基本原则

* 在程序运行时，同一个对象多次调用hashCode()方法应该返回相同的值
* 当两个对象的equals()方法比较返回true时，这两个对象的hashCode()方法的返回值也应相等
* 对象中用作equals()方法比较的Field，都应该用来计算hashCode值

1. 重写equals()方法的基本原则

以自定义的Customer类为例，何时需要重写equals()？ 

* 当一个类有自己特有的“逻辑相等”概念,当改写equals()的时候，总是要改写hashCode()，根据一个类的equals方法（改写后），两个截然不同的实例有可能在逻辑上是相等的，但是，根据Object.hashCode()方法，它们仅仅是两个对象。
* 因此，违反了“相等的对象必须具有相等的散列码”。
* 结论：复写equals方法的时候一般都需要同时复写hashCode方法。通常参与计算hashCode的对象的属性也应该参与到equals()中进行计算。

### Set实现类之二：LinkedHashSet

1. LinkedHashSet是HashSet的子类，在添加数据的同时，每个数据还维护了两个引用，记录此数据前一个数据和后一个数据
2. LinkedHashSet根据元素的hashCode值来决定元素的存储位置，但它同时使用双向链表维护元素的次序，这使得元素看起来是以插入顺序保存的
3. LinkedHashSet插入性能略低于HashSet，但在迭代访问Set里的全部元素时有很好的性能。
4. LinkedHashSet不允许集合元素重复
5. 对于频繁的遍历操作，LinkedHashSet效率高于HashSet

### Set实现类之三：TreeSet

1）TreeSet概述

* TreeSet是SortedSet接口的实现类，TreeSet可以确保集合元素处于排序状态。
* TreeSet底层使用红黑树结构存储数据（TreeMap相同），特点是有序，查询速度比List快
* TreeSet两种排序方法：自然排序（实现Comparable接口）和定制排序（Comparator）。默认情况下，TreeSet采用自然排序。
* 向TreeSet中添加的数据，要求是相同类的对象

reeSet set = **new** TreeSet();  
  
*//失败：不能添加不同类的对象*set.add(123);  
set.add(456);  
set.add(**"AA"**);  
set.add(**new** User(**"Tom"**,12));

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

2）排 序—自然排序

* 自然排序：TreeSet会调用集合元素的compareTo(Object obj)方法来比较元素之间的大小关系，然后将集合元素按升序(默认情况)排列
* 如果试图把一个对象添加到TreeSet时，则该对象的类必须实现Comparable接口。实现Comparable的类必须实现compareTo(Object obj)方法，两个对象即通过compareTo(Object obj) 方法的返回值来比较大小 
* Comparable的典型实现：
* BigDecimal、BigInteger以及所有的数值型对应的包装类：按它们对应的数值大小进行比较
* Character：按字符的unicode值来进行比较
* Boolean：true 对应的包装类实例大于false对应的包装类实例
* String：按字符串中字符的unicode值进行比较
* Date、Time：后边的时间、日期比前面的时间、日期大
* 向TreeSet中添加元素时，只有第一个元素无须比较compareTo()方法，后面添加的所有元素都会调用compareTo()方法进行比较。
* 因为只有相同类的两个实例才会比较大小，所以向TreeSet中添加的应该是同一个类的对象。
* 对于TreeSet集合而言，它判断两个对象是否相等的唯一标准是：两个对象通过compareTo(Object obj)方法比较返回值是否为0，不再是equals()
* 当需要把一个对象放入TreeSet中，重写该对象对应的equals()方法时，应保证该方法与 compareTo(Object obj)方法有一致的结果：如果两个对象通过equals()方法比较返回true，则通过compareTo(Object obj)方法比较应返回0。否则，让人难以理解

*//实现Comparable接口，重写compareTo方法。按照姓名从大到小排列,年龄从小到大排列*@Override  
**public int** compareTo(Object o) {  
 **if**(o **instanceof** User){  
 User user = (User)o;  
*// return -this.name.compareTo(user.name);* **int** compare = -**this**.**name**.compareTo(user.**name**); //先比较姓名  
 **if**(compare != 0){  
 **return** compare;  
 }**else**{  
 **return** Integer.*compare*(**this**.**age**,user.**age**); //再比较年龄  
 }  
 }**else**{  
 **throw new** RuntimeException(**"输入的类型不匹配"**);  
}}

3）排 序—定制排序

* TreeSet的自然排序要求元素所属的类实现Comparable接口，如果元素所属的类没有实现Comparable接口，或不希望按照升序(默认情况)的方式排列元素或希望按照其它属性大小进行排序，则考虑使用定制排序。
* 定制排序，通过Comparator接口来实现。需要重写compare(T o1,T o2)方法
* 利用int compare(T o1,T o2)方法，比较o1和o2的大小：如果方法返回正整数，则表示o1大于o2；如果返回0，表示相等；返回负整数，表示o1小于o2 
* 要实现定制排序，需要将实现Comparator接口的实例作为形参传递给TreeSet的构造器
* 此时，仍然只能向TreeSet中添加类型相同的对象。否则发生ClassCastException异常
* 使用定制排序判断两个元素相等的标准是：通过Comparator比较两个元素返回了0

Comparator com = **new** Comparator() {  
 *//按照年龄从小到大排列，年龄相同则保留一条* @Override  
 **public int** compare(Object o1, Object o2) {  
 **if**(o1 **instanceof** User && o2 **instanceof** User){  
 User u1 = (User)o1;  
 User u2 = (User)o2;  
 **return** Integer.*compare*(u1.getAge(),u2.getAge());  
 }**else**{  
 **throw new** RuntimeException(**"输入的数据类型不匹配"**);  
}}};  
TreeSet set = **new** TreeSet(com);  
set.add(**new** User(**"Tom"**,12));  
set.add(**new** User(**"Jerry"**,32));  
set.add(**new** User(**"Jim"**,2));  
set.add(**new** User(**"Mike"**,65));  
set.add(**new** User(**"Mary"**,33));  
set.add(**new** User(**"Jack"**,33));  
set.add(**new** User(**"Jack"**,56));  
  
Iterator iterator = set.iterator();  
**while**(iterator.hasNext()){  
 System.***out***.println(iterator.next());  
}

## Map接口

### Map接口概述

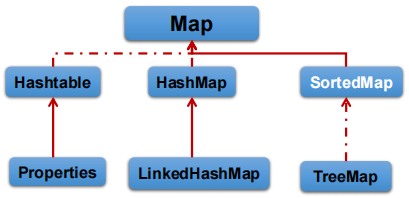
1. Map与Collection并列存在。双列数据，用于保存具有映射关系的数据：key-value；一个键值对：key-value构成了一个Entry对象
2. Map中的key和value都可以是任何引用类型的数据
3. Map中的key：无序的、不可重复的，使用Set存储所有的key，即同一个Map对象所对应的类，须重写hashCode()和equals()方法；常用String类作为Map的“键”
4. Map中的value：无序的、可重复的，使用Collection存储所有的value。value所在的类要重写equals()
5. key和value之间存在单向一对一关系，即通过指定的key总能找到唯一的、确定的value
6. Map接口的常用实现类：HashMap、TreeMap、LinkedHashMap和Properties

* HashMap：作为Map的主要实现类；线程不安全的，效率高；存储null的key和value
* LinkedHashMap：HashMap的子类，保证在遍历map元素时，可以按照添加的顺序实现遍历。

原因：在原有的HashMap底层结构基础上，添加了一对指针，指向前一个和后一个元素

对于频繁的遍历操作，此类执行效率高于HashMap

* TreeMap：保证按照添加的key-value对进行排序，实现排序遍历。此时考虑key的自然排序或定制排序。底层使用红黑树
* Hashtable：作为古老的实现类；线程安全的，效率低；不能存储null的key和value
* Properties：Hashtable的子类，常用来处理配置文件。key和value都是String类型



### Map接口：常用方法

1. 添加、删除、修改操作

* Object put(Object key,Object value)：将指定key-value添加到(或修改)当前map对象中
* void putAll(Map m)：将m中的所有key-value对存放到当前map中
* Object remove(Object key)：移除指定key的key-value对，并返回value
* void clear()：清空当前map中的所有数据

Map map = **new** HashMap();  
*//添加*map.put(**"AA"**,123);  
map.put(45,123);  
map.put(**"BB"**,56);  
*//修改*map.put(**"AA"**,87);  
System.***out***.println(map); *//{AA=87, BB=56, 45=123}*Map map1 = **new** HashMap();  
map1.put(**"CC"**,123);  
map1.put(**"DD"**,123);

*//将map1中的所有key-value对存放到当前map中*  
map.putAll(map1);  
System.***out***.println(map); *//{AA=87, BB=56, CC=123, DD=123, 45=123}  
  
//remove(Object key)*Object value = map.remove(**"CC"**);  
System.***out***.println(value); *//123*System.***out***.println(map); *//{AA=87, BB=56, DD=123, 45=123}  
  
//clear()*map.clear();*//与map = null操作不同*System.***out***.println(map.size()); *//0*System.***out***.println(map); *//{}*

1. 元素查询的操作

* Object get(Object key)：获取指定key对应的value
* boolean containsKey(Object key)：是否包含指定的key
* boolean containsValue(Object value)：是否包含指定的value
* int size()：返回map中key-value对的个数
* boolean isEmpty()：判断当前map是否为空
* boolean equals(Object obj)：判断当前map和参数对象obj是否相等

Map map = **new** HashMap();  
map.put(**"AA"**,123);  
map.put(45,123);  
map.put(**"BB"**,56);System.***out***.println(map.get(45)); *//123***boolean** isExist = map.containsKey(**"BB"**);  
System.***out***.println(isExist); *//true*isExist = map.containsValue(123);  
System.***out***.println(isExist); *//true*map.clear();  
System.***out***.println(map.isEmpty()); *//true*

1. 元视图操作的方法

* Set keySet()：返回所有key构成的Set集合
* Collection values()：返回所有value构成的Collection集合
* Set entrySet()：返回所有key-value对构成的Set集合

Map map = **new** HashMap();  
map.put(**"AA"**,123);  
map.put(45,1234);  
map.put(**"BB"**,56);  
*//遍历所有的key集：keySet()*Set set = map.keySet();  
Iterator iterator = set.iterator();  
**while**(iterator.hasNext()){  
 System.***out***.println(iterator.next());  
}  
*//遍历所有的value集：values()*Collection values = map.values();  
**for**(Object obj : values){  
 System.***out***.println(obj);  
}  
*//遍历所有的key-value  
//方式一：entrySet()*Set entrySet = map.entrySet();  
Iterator iterator1 = entrySet.iterator();  
**while**(iterator1.hasNext()){  
 Object obj = iterator1.next();  
 *//entrySet集合中的元素都是entry（每个键值对）* Map.Entry entry = (Map.Entry) obj;  
 System.***out***.println(entry.getKey() + **"---->"** + entry.getValue());  
}  
*//方式二：*Set keySet = map.keySet();  
Iterator iterator2 = keySet.iterator();  
**while**(iterator2.hasNext()){  
 Object key = iterator2.next();  
 Object value = map.get(key);  
 System.***out***.println(key + **"====="** + value);  
}

1. 总结：常用方法

* 添加：put(Object key,Object value)
* 删除：remove(Object key)
* 修改：put(Object key,Object value)
* 查询：get(Object key)
* 长度：size()
* 遍历：keySet() / values() / entrySet()

### Map实现类之一：HashMap

1. 一个key-value构成一个entry
2. 所有的entry构成的集合是Set:无序的、不可重复的
3. HashMap判断两个key相等的标准是：两个key通过equals()方法返回true，hashCode值也相等
4. HashMap判断两个value相等的标准是：两个value通过equals()方法返回true
5. 面试题：HashMap的底层实现原理？

### Map实现类之二：LinkedHashMap

1. LinkedHashMap是HashMap的子类
2. 在HashMap存储结构的基础上，使用了一对双向链表来记录添加元素的顺序
3. 与LinkedHashSet类似，LinkedHashMap可以维护Map的迭代顺序：迭代顺序与Key-Value对的插入顺序一致

### Map实现类之三：TreeMap

1. TreeMap存储Key-Value对时，需要根据key-value对进行排序。TreeMap可以保证所有的 Key-Value对处于有序状态。 
2. TreeSet底层使用红黑树结构存储数据
3. TreeMap的Key的排序：

* 自然排序：TreeMap 的所有的 Key 必须实现 Comparable 接口，而且所有的 Key 应该是同一个类的对象，否则将会抛出 ClasssCastException
* 定制排序：创建 TreeMap 时，传入一个 Comparator 对象，该对象负责对TreeMap 中的所有 key 进行排序。此时不需要 Map 的 Key 实现Comparable 接口

1. TreeMap判断两个key相等的标准：两个key通过compareTo()方法或者compare()方法返回0。

### Map实现类之四：Hashtable

1. Hashtable是个古老的Map实现类，JDK1.0就提供了。不同于HashMap，Hashtable是线程安全的
2. Hashtable实现原理和HashMap相同，功能相同。底层都使用哈希表结构，查询速度快，很多情况下可以互用。 
3. 与HashMap不同，Hashtable不允许使用null作为key和value
4. 与HashMap一样，Hashtable也不能保证其中Key-Value对的顺序
5. Hashtable判断两个key相等、两个value相等的标准，与HashMap一致

### Map实现类之五：Properties

1. Properties类是Hashtable的子类，该对象用于处理属性文件
2. 由于属性文件里的key、value都是字符串类型，所以Properties里的key和value都是字符串类型
3. 存取数据时，建议使用setProperty(String key,String value)方法和getProperty(String key)方法

FileInputStream fis = **null**;  
**try** {  
 Properties pros = **new** Properties();  
 fis = **new** FileInputStream(**"jdbc.properties"**);  
 pros.load(fis); *//加载流对应的文件* String name = pros.getProperty(**"name"**);  
 String password = pros.getProperty(**"password"**);  
 System.***out***.println(**"name = "** + name + **", password = "** + password);  
} **catch** (IOException e) {  
 e.printStackTrace();  
} **finally** {  
 **if**(fis != **null**){  
 **try** {  
 fis.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
}}}

## Collections工具类

### Collections工具类概述

1. Collections 是一个操作 Set、List 和 Map 等集合的工具类
2. Collections 中提供了一系列静态的方法对集合元素进行排序、查询和修改等操作，还提供了对集合对象设置不可变、对集合对象实现同步控制等方法

### Collections常用方法

1. 排序操作：（均为static方法）

* reverse(List)：反转List中元素的顺序
* shuffle(List)：对List集合元素进行随机排序
* sort(List)：根据元素的自然顺序对指定List集合元素按升序排序
* sort(List，Comparator)：根据指定的Comparator产生的顺序对List集合元素进行排序
* swap(List，int， int)：将指定list集合中的 i 处元素和 j 处元素进行交换

List list = **new** ArrayList();  
list.add(123);  
list.add(43);  
list.add(765);  
list.add(765);  
list.add(765);  
list.add(-97);  
list.add(0);  
System.***out***.println(list); *//[123, 43, 765, 765, 765, -97, 0]*Collections.*reverse*(list); *//[0, -97, 765, 765, 765, 43, 123]  
//Collections.shuffle(list); //随机排序  
//Collections.sort(list); //[-97, 0, 43, 123, 765, 765, 765]  
//Collections.swap(list,1,2); //[123, 765, 43, 765, 765, -97, 0]  
//int frequency = Collections.frequency(list, 765); //3*System.***out***.println(list);

2）查找、替换

* Object max(Collection)：根据元素的自然顺序，返回给定集合中的最大元素
* Object max(Collection，Comparator)：根据Comparator指定的顺序，返回给定集合中的最大元素
* Object min(Collection)
* Object min(Collection，Comparator)
* int frequency(Collection，Object)：返回指定集合中指定元素的出现次数
* void copy(List dest,List src)：将src中的内容复制到dest中
* boolean replaceAll(List list，Object oldVal，Object newVal)：使用新值替换 List 对象的所有旧值

List list = **new** ArrayList();  
list.add(123);  
list.add(43);  
list.add(765);  
list.add(-97);  
list.add(0);  
  
*//报异常：IndexOutOfBoundsException("Source does not fit in dest")  
//List dest = new ArrayList();  
//Collections.copy(dest,list);  
//正确的：*List dest = Arrays.*asList*(**new** Object[list.size()]);  
System.***out***.println(dest.size()); *//list.size();*Collections.*copy*(dest,list);  
  
System.***out***.println(dest);

3）同步控制

Collections类中提供了多个synchronizedXxx()方法，该方法可使将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题。

*//返回的list1即为线程安全的List*List list1 = Collections.*synchronizedList*(list);

## 数据结构简述

# 第13章 IO流

## File类的使用

### 基本介绍

1）File类的一个对象，代表一个文件或一个文件目录

2）File类声明在java.io包下

3）File类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小等方法，并未涉及到写入或读取文件内容的操作。如果需要读取或写入文件内容，必须使用IO流来完成。

4）后续File类的对象常会作为参数传递到流的构造器中，指明读取或写入的"终点"

### 创建File类的实例

1）File(String filePath)

2）File(String parentPath,String childPath)

以parent为父路径，child为子路径创建File对象

3）File(File parentFile,String childPath)

根据一个父File对象和子文件路径创建File对象

相对路径：相较于某个路径下，指明的路径

绝对路径：包含盘符在内的文件或文件目录的路径

注：File不会判断文件或目录是否存在，只是加载至内存，输出对应的文件路径

//构造器1  
File file1 = new File("hello.txt");//相对于当前module  
File file2 = new File("D:\\workspace\_idea1\\JavaSenior\\day08\\he.txt");  
  
System.out.println(file1); //hello.txt  
System.out.println(file2); //D:\workspace\_idea1\JavaSenior\day08\he.txt  
  
//构造器2：  
File file3 = new File("D:\\workspace\_idea1","JavaSenior");  
System.out.println(file3); //D:\workspace\_idea1\JavaSenior  
  
//构造器3：  
File file4 = new File(file3,"hi.txt");   
System.out.println(file4); //D:\workspace\_idea1\JavaSenior\hi.txt

### 常用方法一：获取功能

1）public String getAbsolutePath()：获取绝对路径

2）public String getPath() ：获取路径

3）public String getName() ：获取名称

4）public String getParent()：获取上层文件目录路径。若无，返回null（相对路径返回null）

5）public long length() ：获取文件长度（即：字节数）。不能获取目录的长度。

6）public long lastModified() ：获取最后一次的修改时间，毫秒值

如下的两个方法适用于文件目录：

7）public String[] list() ：获取指定目录下的所有文件或者文件目录的名称数组

8）public File[] listFiles() ：获取指定目录下的所有文件或者文件目录的File数组

File file = new File("E:\\test");

String[] list = file.list();

for(String s : list){

System.out.println(s); //test1 test2.txt

}

File[] files = file.listFiles();

for(File f : files){

System.out.println(f); //E:\test\test1 E:\test\test2.txt

}

9）public boolean renameTo(File dest):把文件重命名为指定的文件路径

比如：file1.renameTo(file2)为例。要想保证返回true,需要file1在硬盘中是存在的，且file2不能在硬盘中存在。

File file1 = new File("hello.txt");

File file2 = new File("D:\\io\\hi.txt");

boolean renameTo = file2.renameTo(file1);

System.out.println(renameTo);

### 常用方法二：判断功能

public boolean isDirectory()：判断是否是文件目录

public boolean isFile() ：判断是否是文件

public boolean exists() ：判断是否存在

public boolean canRead() ：判断是否可读

public boolean canWrite() ：判断是否可写

public boolean isHidden() ：判断是否隐藏

### 常用方法二：创建与删除

**1）创建文件**

public boolean createNewFile() ：创建文件。若文件存在，则不创建，返回false

File file1 = new File("hi.txt");

if(!file1.exists()){

//文件的创建

file1.createNewFile();

System.out.println("创建成功");

}else{ //文件存在

file1.delete();

System.out.println("删除成功");

}

**2）创建文件目录**

public boolean mkdir()：创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建

public boolean mkdirs()：创建文件目录。如果此文件目录存在，就不创建了。如果上层文件目录不存在，一并创建

注意事项：如果你创建文件或者文件目录没有写盘符路径，那么，默认在项目路径下。

//文件目录的创建

File file1 = new File("d:\\io\\io1\\io3");

boolean mkdir = file1.mkdir();

if(mkdir){

System.out.println("创建成功1");

}

File file2 = new File("d:\\io\\io1\\io4");

boolean mkdir1 = file2.mkdirs();

if(mkdir1){

System.out.println("创建成功2");

}

**3）删除磁盘中的文件或文件目录**

public boolean delete()：删除文件或者文件夹

删除注意事项：Java中的删除不走回收站

//要想删除成功，io4文件目录下不能有子目录或文件

File file3 = new File("D:\\io\\io1\\io4");

file3 = new File("D:\\io\\io1");

System.out.println(file3.delete()); //false

### File类的使用

File dir1 = new File("D:/IOTest/dir1");

if (!dir1.exists()) { //如果D:/IOTest/dir1不存在，就创建为目录

dir1.mkdir();

}

//创建以dir1为父目录,名为"dir2"的File对象

File dir2 = new File(dir1, "dir2");

if (!dir2.exists()) { //如果还不存在，就创建为目录

dir2.mkdirs();

}

File dir4 = new File(dir1, "dir3/dir4");

if (!dir4.exists()) {

dir4.mkdirs();

}

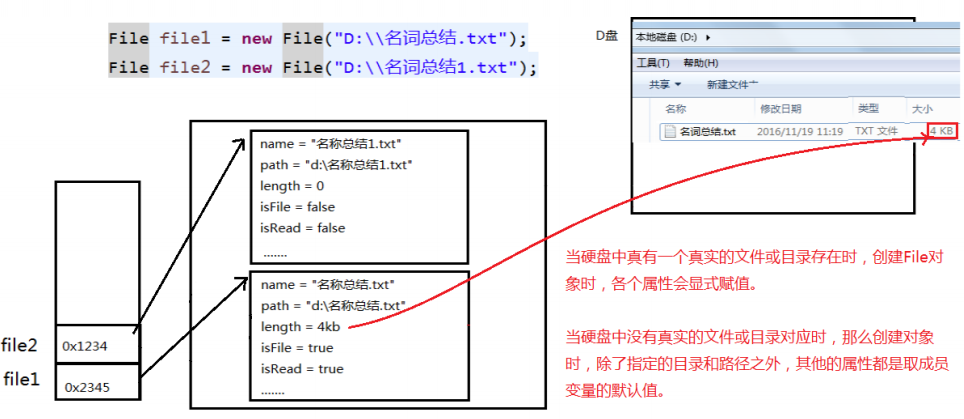
//创建以dir2为父目录,名为"test.txt"的File对象

File file = new File(dir2, "test.txt");

if (!file.exists()) { //如果还不存在，就创建为文件

file.createNewFile();

}



### 练习题

## 二、 IO流原理及流的分类

### Java IO原理

1）I/O是Input/Output的缩写，I/O技术是非常实用的技术，用于处理设备之间的数据传输。如读/写文件，网络通讯等。

2）Java程序中，对于数据的输入/输出操作以“流(stream)”的方式进行。

3）java.io包下提供了各种“流”类和接口，用以获取不同种类的数据，并通过标准的方法输入或输出数据。

4）输入input：读取外部数据（磁盘、光盘等存储设备的数据）到程序（内存）中。

5）输出output：将程序（内存）数据输出到磁盘、光盘等存储设备中

### 流的分类

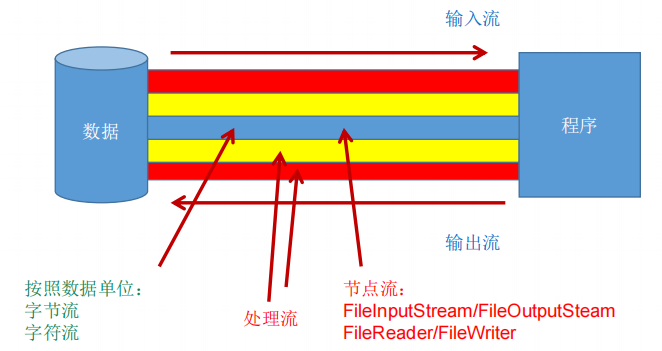
1）按操作数据单位不同分为：字节流(8 bit)（图片、视频等非文本），字符流(16 bit)（文本）

2）按数据流的流向不同分为：输入流，输出流

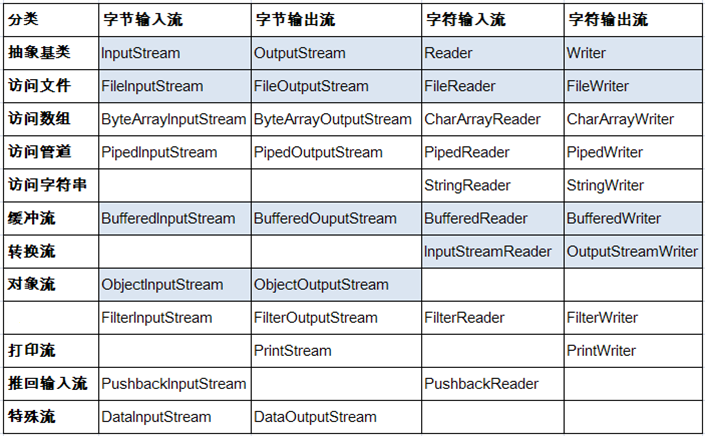
3）按流的角色的不同分为：节点流，处理流

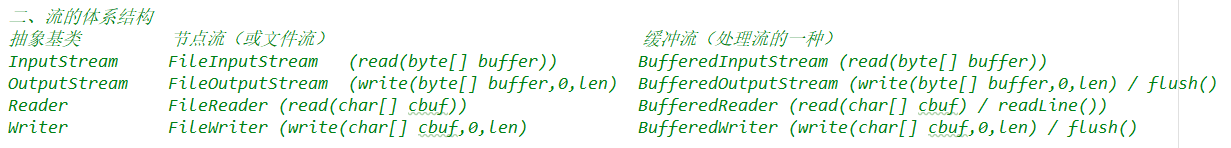
·节点流：直接从数据源或目的地读写数据

·处理流：不直接连接到数据源或目的地，而是“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能



### IO流体系





## 三、文件流的使用

文件流：属于节点流

FileInputStream

FileOutputStream

FileReader

FileWriter

### 1、FileReader、FileWriter访问文本文件

一、FileReader读取数据：

FileReader fr = null;

try {

//1.实例化File类的对象，指明要操作的文件

File file = new File("hello.txt");//相较于当前Module

//2.提供具体的流

fr = new FileReader(file);

//3.数据的读入

//read()：从输入流中读取数据的下一个字节。返回0到255范围内的int字节值。如果达到文件末尾，返回-1

int data;

while((data = fr.read()) != -1){

System.out.print((char)data);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.流的关闭操作

if(fr != null){

try {

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

说明点：

1. read()的理解：返回读入的一个字符。如果达到文件末尾，返回-1

2. 异常的处理：为了保证流资源一定可以执行关闭操作。需要使用try-catch-finally处理

3. 读入的文件一定要存在，否则就会报FileNotFoundException。

对read()操作升级：使用read的重载方法

FileReader fr = null;

try {

//1.File类的实例化

File file = new File("hello.txt");

//2.FileReader流的实例化

fr = new FileReader(file);

//3.读入的操作

//read(char[] cbuf):从此输入流中将最多b.length个字节的数据读入一个byte数组中。如果达到文件末尾，返回-1

char[] cbuf = new char[5];

int len;

while((len = fr.read(cbuf)) != -1){

//方式一：

//错误的写法

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

// System.out.print(cbuf[i]); //输出：helloworld123ld

// }

//正确的写法

// for(int i = 0;i < len;i++){

// System.out.print(cbuf[i]); //输出：helloworld123

// }

//方式二：

//错误的写法,对应着方式一的错误的写法

// String str = new String(cbuf);

// System.out.print(str);

//正确的写法

String str = new String(cbuf,0,len);

System.out.print(str);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

if(fr != null){

//4.资源的关闭

try {

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

二、FileWriter写入数据：

FileWriter fw = null;

try {

//1.提供File类的对象，指明写出到的文件

File file = new File("hello1.txt");

//2.提供FileWriter的对象，用于数据的写出

fw = new FileWriter(file,false);

//3.写出的操作

fw.write("I have a dream!\n");

fw.write("you need to have a dream!");

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.流资源的关闭

if(fw != null){

try {

fw.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

说明：

1. 输出操作，对应的File可以不存在的。并不会报异常

2.File对应的硬盘中的文件如果不存在，在输出的过程中，会自动创建此文件。

3.File对应的硬盘中的文件如果存在：

如果流使用的构造器是：FileWriter(file,false) / FileWriter(file):对原有文件的覆盖

如果流使用的构造器是：FileWriter(file,true):不会对原有文件覆盖，而是在原有文件基础上追加内容

三、文本文件的复制：

FileReader fr = null;

FileWriter fw = null;

try {

//1.创建File类的对象，指明读入和写出的文件

File srcFile = new File("hello.txt");

File destFile = new File("hello2.txt");

//不能使用字符流来处理图片等字节数据

// File srcFile = new File("爱情与友情.jpg");

// File destFile = new File("爱情与友情1.jpg");

//2.创建输入流和输出流的对象

fr = new FileReader(srcFile);

fw = new FileWriter(destFile);

//3.数据的读入和写出操作

char[] cbuf = new char[5];

int len; //记录每次读入到cbuf数组中的字符的个数

while((len = fr.read(cbuf)) != -1){

//每次写出len个字符

fw.write(cbuf,0,len);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.关闭流资源

try {

if(fw != null)

fw.close();

} catch (IOException e) {

e.printStackTrace();

}

try {

if(fr != null)

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

### 2、FileInputStream、FileOutputStream访问非文本文件

FileInputStream fis = null;

FileOutputStream fos = null;

try {

//1. 造文件

File srcFile = new File("爱情与友情.jpg");

File destFile = new File("爱情与友情2.jpg");

//2.造流

fis = new FileInputStream(srcFile);

fos = new FileOutputStream(destFile);

//3.复制的过程

byte[] buffer = new byte[5];

int len;

while((len = fis.read(buffer)) != -1){

fos.write(buffer,0,len);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.关闭资源

fos.close();

fis.close();

}

## 四、缓冲流的使用

1）缓冲流：处理流的一种

BufferedInputStream

BufferedOutputStream

BufferedReader

BufferedWriter

2）作用：提供流的读取、写入的速度

提高读写速度的原因：内部提供了一个缓冲区

3）处理流，就是“套接”在已有的流（节点流/处理流）的基础上

4）实现非文本文件的复制：

BufferedInputStream bis = null;

BufferedOutputStream bos = null;

try {

//1.造文件

File srcFile = new File("爱情与友情.jpg");

File destFile = new File("爱情与友情3.jpg");

//2.造流

//2.1 造节点流

FileInputStream fis = new FileInputStream((srcFile));

FileOutputStream fos = new FileOutputStream(destFile);

//2.2 造缓冲流

bis = new BufferedInputStream(fis);

bos = new BufferedOutputStream(fos);

//3.复制的细节：读取、写入

byte[] buffer = new byte[10]; //创建一个临时存放数据的数组

int len;

while((len = bis.read(buffer)) != -1){

bos.write(buffer,0,len); //write的同时会自动刷新缓冲区

// bos.flush(); //刷新缓冲区

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.资源关闭

//要求：先关闭外层的流，再关闭内层的流

if(bos != null){

try {

bos.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if(bis != null){

try {

bis.close();

} catch (IOException e) {

e.printStackTrace();

}

}

//说明：关闭外层流的同时，内层流也会自动的进行关闭。关于内层流的关闭，我们可以省略.

// fos.close();

// fis.close();

}

5）使用BufferedReader和BufferedWriter实现文本文件的复制

BufferedReader br = null;

BufferedWriter bw = null;

try {

**//创建文件和相应的流**

br = new BufferedReader(new FileReader(new File("dbcp.txt")));

bw = new BufferedWriter(new FileWriter(new File("dbcp1.txt")));

**//读写操作**

//方式一：使用char[]数组

// char[] cbuf = new char[1024];

// int len;

// while((len = br.read(cbuf)) != -1){

// bw.write(cbuf,0,len);

// //bw.flush();

// }

//方式二：使用String

String data;

while((data = br.readLine()) != null){

//方法一：

// bw.write(data + "\n"); //data中不包含换行符

//方法二：

bw.write(data); //data中不包含换行符

bw.newLine(); //提供换行的操作

}

} catch (IOException e) {

e.printStackTrace();

} finally {

**//关闭资源**

bw.close();

br.close();

}

6）练习题：图片的加解密

FileInputStream fis = null;

FileOutputStream fos = null;

try {

fis = new FileInputStream("爱情与友情.jpg");

fos = new FileOutputStream("爱情与友情secret1.jpg");

byte[] buffer = new byte[20];

int len;

while ((len = fis.read(buffer)) != -1) {

//字节数组进行修改

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

buffer[i] = (byte) (buffer[i] ^ 5);

}

fos.write(buffer, 0, len);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.关流

fos.close();

fis.close();

}

6）练习题：获取文本上字符出现的次数,把数据写入文件

FileReader fr = null;

BufferedWriter bw = null;

try {

//1.创建Map集合

Map<Character, Integer> map = new HashMap<Character, Integer>();

//2.遍历每一个字符,每一个字符出现的次数放到map中

fr = new FileReader("dbcp.txt");

int c = 0;

while ((c = fr.read()) != -1) {

//int 还原 char

char ch = (char) c;

//判断char是否在map中第一次出现

if (map.get(ch) == null) {

map.put(ch, 1);

} else {

map.put(ch, map.get(ch) + 1);

}

}

//3.把map中数据存在文件count.txt

//3.1 创建Writer

bw = new BufferedWriter(new FileWriter("wordcount.txt"));

//3.2 遍历map,再写入数据

Set<Map.Entry<Character, Integer>> entrySet = map.entrySet();

for (Map.Entry<Character, Integer> entry : entrySet) {

switch (entry.getKey()) {

case ' ':

bw.write("空格=" + entry.getValue());

break;

case '\t':

bw.write("tab键=" + entry.getValue());

break;

case '\r':

bw.write("回车=" + entry.getValue());

break;

case '\n':

bw.write("换行=" + entry.getValue());

break;

default:

bw.write(entry.getKey() + "=" + entry.getValue());

break;

}

bw.newLine();

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//4.关流

fr.close();

bw.close();

}

## 五、转换流的使用

1）转换流：属于字符流、处理流

InputStreamReader：将一个字节的输入流转换为字符的输入流

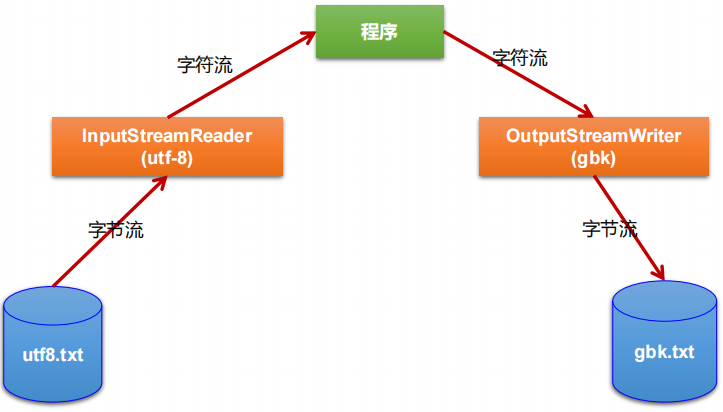
OutputStreamWriter：将一个字符的输出流转换为字节的输出流

2）作用：提供字节流与字符流之间的转换。很多时候我们使用转换流来处理文件乱码问题。实现编码和

解码的功能

3）解码：字节、字节数组 ---> 字符数组、字符串

编码：字符数组、字符串 ---> 字节、字节数组



4）字符集

ASCII：美国标准信息交换码。用一个字节的7位可以表示

ISO8859-1：拉丁码表。欧洲码表。用一个字节的8位表示

GB2312：中国的中文编码表。最多两个字节编码所有字符

GBK：中国的中文编码表升级，融合了更多的中文文字符号。最多两个字节编码

Unicode：国际标准码，融合了目前人类使用的所有字符。为每个字符分配唯一的字符码。所有的文字都用两个字节来表示

UTF-8：变长的编码方式，可用1-4个字节来表示一个字符。

5）InputStreamReader的使用，实现字节的输入流到字符的输入流的转换

FileInputStream fis = new FileInputStream("dbcp.txt");

//InputStreamReader isr = new InputStreamReader(fis); //使用系统默认的字符集

//参数2指明了字符集，具体使用哪个字符集，取决于文件dbcp.txt保存时使用的字符集

InputStreamReader isr = new InputStreamReader(fis,"UTF-8");

char[] cbuf = new char[20];

int len;

while((len = isr.read(cbuf)) != -1){

String str = new String(cbuf,0,len);

System.out.print(str);

}

isr.close();

6）综合使用InputStreamReader和OutputStreamWriter

//1.造文件、造流

File file1 = new File("dbcp.txt");

File file2 = new File("dbcp\_gbk.txt");

FileInputStream fis = new FileInputStream(file1);

FileOutputStream fos = new FileOutputStream(file2);

InputStreamReader isr = new InputStreamReader(fis,"utf-8");

OutputStreamWriter osw = new OutputStreamWriter(fos,"gbk");

//2.读写过程

char[] cbuf = new char[20];

int len;

while((len = isr.read(cbuf)) != -1){

osw.write(cbuf,0,len);

}

//3.关闭资源

isr.close();

osw.close();

## 六、标准输入、输出流（了解）

1）System.in：标准的输入流，默认从键盘输入 （字节流）

System.out：标准的输出流，默认从控制台输出

2）System类的setIn(InputStream is) / setOut(PrintStream ps)方式重新指定输入和输出的流

3）练习：

从键盘输入字符串，要求将读取到的整行字符串转成大写输出。然后继续进行输入操作，直至当输入“e”或者“exit”时，退出程序。

方法一：使用Scanner实现，调用next()返回一个字符串

方法二：使用System.in实现。System.in ---> 转换流 ---> BufferedReader的readLine()

BufferedReader br = null;

try {

InputStreamReader isr = new InputStreamReader(System.in);

br = new BufferedReader(isr);

while (true) {

System.out.println("请输入字符串：");

String data = br.readLine();

if ("e".equalsIgnoreCase(data) || "exit".equalsIgnoreCase(data)) {

System.out.println("程序结束");

break;

}

String upperCase = data.toUpperCase();

System.out.println(upperCase);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

br.close();

}

## 七、打印流（了解）

1）实现将基本数据类型的数据格式转化为字符串输出

2）打印流：PrintStream和PrintWriter

* 提供了一系列重载的print()和println()方法，用于多种数据类型的输出
* PrintStream和PrintWriter的输出不会抛出IOException异常
* PrintStream和PrintWriter有自动flush功能
* PrintStream 打印的所有字符都使用平台的默认字符编码转换为字节。在需要写入字符而不是写入字节的情况下，应该使用 PrintWriter 类。
* System.out返回的是PrintStream的实例

PrintStream ps = null;

try {

FileOutputStream fos = new FileOutputStream(new File("D:\\IO\\text.txt"));

//创建打印输出流,设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新输出缓冲区)

ps = new PrintStream(fos, true);

if (ps != null) { //把标准输出流(控制台输出)改成文件

System.setOut(ps);

}

for (int i = 0; i <= 255; i++) { //输出ASCII字符

System.out.print((char) i);

if (i % 50 == 0) { //每50个数据一行

System.out.println(); //换行

}

}

} catch (FileNotFoundException e) {

e.printStackTrace();

} finally {

ps.close();

}

## 八、数据流（了解）

1）DataInputStream 和 DataOutputStream

2）作用：用于读取或写出基本数据类型的变量或字符串

3）将内存中的字符串、基本数据类型的变量写出到文件中

//1.造文件、造流

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));

//2.写入过程

dos.writeUTF("刘建辰");

dos.flush(); //刷新操作，将内存中的数据写入文件

dos.writeInt(23);

dos.flush();

dos.writeBoolean(true);

dos.flush();

//3.关流

dos.close();

1. 将文件中存储的基本数据类型变量和字符串读取到内存中，保存在变量中
2. 注意点：读取不同类型的数据的顺序要与当初写入文件时，保存的数据的顺序一致！

//1.造文件、造流

DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));

//2.读取过程

String name = dis.readUTF();

int age = dis.readInt();

boolean isMale = dis.readBoolean();

System.out.println("name = " + name);

System.out.println("age = " + age);

System.out.println("isMale = " + isMale);

//3.关流

dis.close();

## 九、对象流

### 基本概念

* ObjectInputStream和OjbectOutputSteam
* 用于存储和读取基本数据类型数据或对象的处理流。它的强大之处就是可以把Java中的对象写入到数据源中，也能把对象从数据源中还原回来
* 序列化：用ObjectOutputStream类保存基本类型数据或对象的机制
* 反序列化：用ObjectInputStream类读取基本类型数据或对象的机制
* ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量

### 对象的序列化

* 对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。//当其它程序获取了这种二进制流，就可以恢复成原来的Java对象
* 序列化的好处在于可将任何实现了Serializable接口的对象转化为字节数据，使其在保存和传输时可被还原
* 如果需要让某个对象支持序列化机制，则必须让对象所属的类及其属性是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一。否则，会抛出NotSerializableException异常Serializable、Externalizable
* 凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态变量：private static final long serialVersionUID;serialVersionUID用来表明类的不同版本间的兼容性。简言之，其目的是以序列化对象进行版本控制，有关各版本反序列化时是否兼容。如果类没有显示定义这个静态常量，它的值是Java运行时环境根据类的内部细节自动生成的。若类的实例变量做了修改，serialVersionUID可能发生变化。故建议，显式声明。
* 简单来说，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。(InvalidCastException)
* 序列化过程：将内存中的java对象保存到磁盘中或通过网络传输出去，使用ObjectOutputStream实现

ObjectOutputStream oos = null;

try {

//1.造文件、造流

oos = new ObjectOutputStream(new FileOutputStream("object.dat"));

//2.写入过程

oos.writeObject(new String("我爱北京天安门"));

oos.flush(); //刷新操作

oos.writeObject(new Person("王铭",23));

oos.flush();

oos.writeObject(new Person("张学良",23,1001,new Account(5000)));

oos.flush();

//3.关流

oos.close();

}

* 反序列化：将磁盘文件中的对象还原为内存中的一个java对象，使用ObjectInputStream来实现

ObjectInputStream ois = null;

try {

ois = new ObjectInputStream(new FileInputStream("object.dat"));

Object obj = ois.readObject();

String str = (String) obj;

Person p = (Person) ois.readObject();

Person p1 = (Person) ois.readObject();

System.out.println(str);

System.out.println(p);

System.out.println(p1);

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} finally {

ois.close();

}

### 自定义类实现序列化

Person需要满足如下的要求，方可序列化

1）需要实现接口：Serializable

2）当前类提供一个全局常量：serialVersionUID

3）除了当前Person类需要实现Serializable接口之外，还必须保证其内部所有属性也必须是可序列化的。（默认情况下，基本数据类型可序列化）

public class Person implements Serializable{

public static final long serialVersionUID = 475463534532L;

private String name;

private int age;

private int id;

private Account acct;

public Person(String name, int age, int id) {

this.name = name;

this.age = age;

this.id = id;

}

public Person(String name, int age, int id, Account acct) {

this.name = name;

this.age = age;

this.id = id;

this.acct = acct;

}

@Override

public String toString() {

return "Person{" +

"name='" + name + '\'' +

", age=" + age +

", id=" + id +

", acct=" + acct +

'}';

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public Person() {

}

}

class Account implements Serializable{

public static final long serialVersionUID = 4754534532L;

private double balance;

@Override

public String toString() {

return "Account{" +

"balance=" + balance +

'}';

}

public double getBalance() {

return balance;

}

public void setBalance(double balance) {

this.balance = balance;

}

public Account(double balance) {

this.balance = balance;

}

}

## 十、随机存取文件流（RandomAccessFile 类）

1、RandomAccessFile声明在java.io包下，但直接继承于java.lang.Object类。并且它实现了DataInput、DataOutput这两个接口，也就意味着这个类既可以读也可以写。

2、RandomAccessFile类支持 “随机访问” 的方式，程序可以直接跳到文件的任意地方来读、写文件。支持只访问文件的部分内容；可以向已存在的文件后追加内容

3、RandomAccessFile对象包含一个记录指针，用以标示当前读写处的位置。RandomAccessFile类对象可以自由移动记录指针：

 long getFilePointer()：获取文件记录指针的当前位置

 void seek(long pos)：将文件记录指针定位到pos位置

4、构造器

 public RandomAccessFile(File file, String mode)

 public RandomAccessFile(String name, String mode) 

5、创建RandomAccessFile类实例需要指定一个mode参数，该参数指定RandomAccessFile的访问模式：

 r: 以只读方式打开

 rw：打开以便读取和写入

 rwd:打开以便读取和写入；同步文件内容的更新

 rws:打开以便读取和写入；同步文件内容和元数据的更新

1. 如果模式为只读r。则不会创建文件，而是会去读取一个已经存在的文件，如果读取的文件不存在则会出现异常
2. 如果模式为rw读写。如果文件不存在则会去创建文件，如果存在则不会创建，并会对原有文件内容进行覆盖（默认情况下，从头覆盖）

RandomAccessFile raf1 = null;

RandomAccessFile raf2 = null;

try {

//1.造文件、造流

raf1 = new RandomAccessFile(new File("爱情与友情.jpg"),"r");

raf2 = new RandomAccessFile(new File("爱情与友情1.jpg"),"rw");

//2.读取与写入过程

byte[] buffer = new byte[1024];

int len;

while((len = raf1.read(buffer)) != -1){

raf2.write(buffer,0,len);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

//3.关流

raf1.close();

raf2.close();

}

1. 使用RandomAccessFile实现数据的插入效果

RandomAccessFile raf1 = new RandomAccessFile("hello.txt","rw");

raf1.seek(3);//将指针调到角标为3的位置

//保存指针3后面的所有数据到StringBuilder中

StringBuilder builder = new StringBuilder((int) new File("hello.txt").length());

byte[] buffer = new byte[20];

int len;

while((len = raf1.read(buffer)) != -1){

builder.append(new String(buffer,0,len)) ;

}

//调回指针，写入“xyz”

raf1.seek(3);

raf1.write("xyz".getBytes());

//将StringBuilder中的数据写入到文件中

raf1.write(builder.toString().getBytes());

raf1.close();

## 补充：使用第三方jar包实现数据读写



import org.apache.commons.io.FileUtils;

import java.io.File;

import java.io.IOException;

File srcFile = new File("day10\\爱情与友情.jpg");

File destFile = new File("day10\\爱情与友情2.jpg");

try {

FileUtils.copyFile(srcFile,destFile);

} catch (IOException e) {

e.printStackTrace();

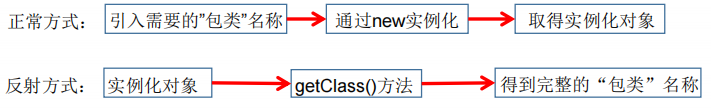
}

# Java反射机制

## Java反射机制概述

### 1、反射概述

1. Reflection（反射）是被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。 
2. 加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：反射。



### 2、Java反射机制提供的功能

1. 在运行时判断任意一个对象所属的类
2. 在运行时构造任意一个类的对象
3. 在运行时判断任意一个类所具有的成员变量和方法
4. 在运行时获取泛型信息
5. 在运行时调用任意一个对象的成员变量和方法
6. 在运行时处理注解
7. 生成动态代理

### 3、反射相关的主要API

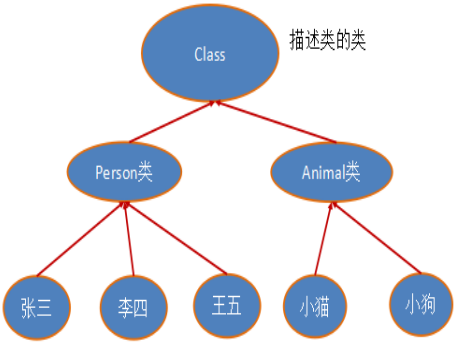
1. java.lang.Class:代表一个类 
2. java.lang.reflect.Method:代表类的方法
3. java.lang.reflect.Field:代表类的成员变量
4. java.lang.reflect.Constructor:代表类的构造器

## 理解Class类并获取Class实例

### 1、Class类概述

对于每个类而言，JRE都为其保留一个不变的Class类型的对象。一个Class对象包含了特定某个结构(class/interface/enum/annotation/primitive type/void/[])的有关信息。 

1. Class本身也是一个类
2. Class 对象只能由系统建立对象
3. 一个加载的类在 JVM 中只会有一个Class实例
4. 一个Class对象对应的是一个加载到JVM中的一个.class文件
5. 每个类的实例都会记得自己是由哪个Class实例所生成
6. 通过Class可以完整地得到一个类中的所有被加载的结构Class类是Reflection的根源，针对任何你想动态加载、运行的类，唯有先获得相应的Class对象



### 2、获取Class类的实例(四种方法)

1）方式一：调用运行时类的属性：.class

前提：若已知具体的类，通过类的class属性获取，该方法最为安全可靠，程序性能最高

实例：Class clazz = String.class;

2）方式二：通过运行时类的对象,调用getClass()

前提：已知某个类的实例，调用该实例的getClass()方法获取Class对象

实例：Class clazz = (new Person()).getClass();

3）方式三：调用Class的静态方法：forName(String classPath)

前提：已知一个类的全类名，且该类在类路径下，可通过Class类的静态方法forName()获取，可能抛出ClassNotFoundException

实例：Class clazz = Class.forName("java.lang.String");

Class clazz1 = Class.forName("com.atguigu.java.Person");

4）方式四：使用类的加载器：ClassLoader(了解)

ClassLoader cl = this.getClass().getClassLoader();

Class clazz4 = cl.loadClass("类的全类名");

*//方式一：调用运行时类的属性：.class*Class clazz1 = Person.**class**;  
System.***out***.println(clazz1); //class com.atguigu.java.Person  
*//方式二：通过运行时类的对象,调用getClass()*Person p1 = **new** Person();  
Class clazz2 = p1.getClass();  
System.***out***.println(clazz2); //class com.atguigu.java.Person  
*//方式三：调用Class的静态方法：forName(String classPath)*Class clazz3 = Class.*forName*(**"com.atguigu.java.Person"**);  
*//Class clazz3 = Class.forName("java.lang.String");*System.***out***.println(clazz3); //class com.atguigu.java.Person  
  
System.***out***.println(clazz1 == clazz2); //true  
System.***out***.println(clazz1 == clazz3); //true

注意：Class的实例就对应着一个运行时类。加载到内存中的运行时类，会缓存一定的时间。在此时间之内，我们可以通过不同的方式来获取此运行时类

### 3、可以有Class对象的类型

（1）class：外部类，成员(成员内部类，静态内部类)，局部内部类，匿名内部类

（2）interface：接口

（3）[]：数组

（4）enum：枚举

（5）annotation：注解@interface

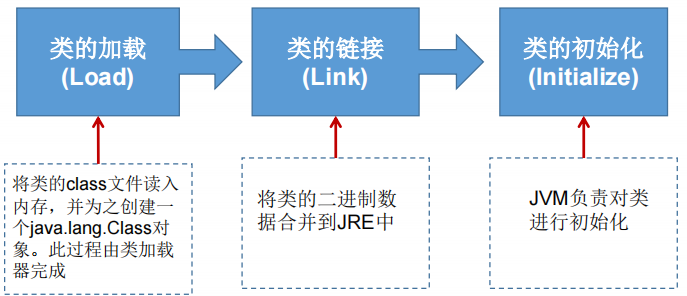
（6）primitive type：基本数据类型

（7）void

## 类的加载与ClassLoader的理解

### 类的加载过程（了解）

程序经过javac.exe命令以后，会生成一个或多个字节码文件(.class结尾)。接着我们使用java.exe命令对某个字节码文件进行解释运行。相当于将某个字节码文件加载到内存中。此过程就称为类的加载。加载到内存中的类，我们就称为运行时类，此运行时类，就作为Class的一个实例



### 使用ClassLoader加载配置文件

Properties pros = **new** Properties(); *//读取配置文件的方式一：*

*//此时的文件默认在当前的module下。*FileInputStream fis = **new** FileInputStream(**"jdbc.properties"**);  
FileInputStream fis1 = **new** FileInputStream(**"src\\jdbc1.properties"**);  
pros.load(fis);  
  
*//读取配置文件的方式二：使用ClassLoader  
//配置文件默认识别为：当前module的src下*ClassLoader classLoader = ClassLoaderTest.**class**.getClassLoader();  
InputStream is = classLoader.getResourceAsStream(**"jdbc1.properties"**);  
pros.load(is);  
  
String user = pros.getProperty(**"user"**);  
String password = pros.getProperty(**"password"**);

## 创建运行时类的对象

1. 创建运行时类的对象的方法：调用Class对象的newInstance()方法

Class<Person> clazz = Person.**class**;Person obj = clazz.newInstance();  
System.***out***.println(obj); *//Person{name='null', age=0}*

Class clazz = Person.**class**;Person obj = (Person)clazz.newInstance();

要 求：

1）类必须有一个无参数的构造器

2）类的构造器的访问权限需要足够，通常，设置为public

**public** Person() {  
 System.***out***.println(**"Person()"**);  
}

1. 在javabean中要求提供一个public的空参构造器。

原因：

1）便于通过反射，创建运行时类的对象

2）便于子类继承此运行时类时，默认调用super()时，保证父类有此构造器

3、体会反射的动态性：运行之后才确定调用的是哪个类

**public void** test2(){  
  
 **for**(**int** i = 0;i < 100;i++){  
 **int** num = **new** Random().nextInt(3);*//0,1,2* String classPath = **""**;  
 **switch**(num){  
 **case** 0:  
 classPath = **"java.util.Date"**;  
 **break**;  
 **case** 1:  
 classPath = **"java.lang.Object"**;  
 **break**;  
 **case** 2:  
 classPath = **"com.atguigu.java.Person"**;  
 **break**;  
 }  
 **try** {  
 Object obj = getInstance(classPath);  
 System.***out***.println(obj);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
}}}  
*/\*  
创建一个指定类的对象。  
classPath:指定类的全类名  
 \*/***public** Object getInstance(String classPath) **throws** Exception {  
 Class clazz = Class.*forName*(classPath);  
 **return** clazz.newInstance();  
}

## 获取运行时类的完整结构

### 1、获取属性及内部结构

Class clazz = Person.**class**;  
 *//getFields():获取当前运行时类及其父类中声明为public访问权限的属性*Field[] fields = clazz.getFields();  
**for**(Field f : fields){  
 System.***out***.println(f);

*//public int com.atguigu.java1.Person.id  
//public double com.atguigu.java1.Creature.weight*  
}  
System.***out***.println(); *//getDeclaredFields():获取当前运行时类中声明的所有属性。（不包含父类中声明的属性）*Field[] declaredFields = clazz.getDeclaredFields();  
**for**(Field f : declaredFields){  
 System.***out***.println(f);

*//private java.lang.String com.atguigu.java1.Person.name  
//int com.atguigu.java1.Person.age  
//public int com.atguigu.java1.Person.id*  
}

*//权限修饰符 数据类型 变量名（了解）*

Class clazz = Person.**class**;  
Field[] declaredFields = clazz.getDeclaredFields();  
**for**(Field f : declaredFields){  
 *//1.权限修饰符* **int** modifier = f.getModifiers();  
 System.***out***.print(Modifier.*toString*(modifier) + **"\t"**); *//private、、public  
  
 //2.数据类型* Class type = f.getType();  
 System.***out***.print(type.getName() + **"\t"**); *//java.lang.String、int、int  
  
 //3.变量名* String fName = f.getName();  
 System.***out***.print(fName); *//name、age、id* System.***out***.println();  
}

### 2、获取方法结构

Class clazz = Person.**class**;  
  
*//getMethods():获取当前运行时类及其所有父类中声明为public权限的方法*Method[] methods = clazz.getMethods();  
**for**(Method m : methods){  
 System.***out***.println(m);  
}  
System.***out***.println();  
*//getDeclaredMethods():获取当前运行时类中声明的所有方法。（不包含父类中声明的方法）*Method[] declaredMethods = clazz.getDeclaredMethods();  
**for**(Method m : declaredMethods){  
 System.***out***.println(m);  
}

### 获取方法的内部结构（了解）

*//权限修饰符 返回值类型 方法名(参数类型1 形参名1,...) throws XxxException{}*

Class clazz = Person.**class**;  
Method[] declaredMethods = clazz.getDeclaredMethods();  
**for**(Method m : declaredMethods){  
 *//1.获取方法声明的注解* Annotation[] annos = m.getAnnotations();  
 **for**(Annotation a : annos){  
 System.***out***.println(a);  
 }  
 *//2.权限修饰符* System.***out***.print(Modifier.*toString*(m.getModifiers()) + **"\t"**);  
  
 *//3.返回值类型* System.***out***.print(m.getReturnType().getName() + **"\t"**);  
  
 *//4.方法名* System.***out***.print(m.getName());  
 System.***out***.print(**"("**);  
 *//5.形参列表* Class[] parameterTypes = m.getParameterTypes();  
 **if**(!(parameterTypes == **null** && parameterTypes.**length** == 0)){  
 **for**(**int** i = 0;i < parameterTypes.**length**;i++){  
 **if**(i == parameterTypes.**length** - 1){  
 System.***out***.print(parameterTypes[i].getName() + **" args\_"** + i);  
 **break**;  
 }  
 System.***out***.print(parameterTypes[i].getName() + **" args\_"** + i + **","**);  
 }  
 }  
 System.***out***.print(**")"**);  
 *//6.抛出的异常* Class[] exceptionTypes = m.getExceptionTypes();  
 **if**(exceptionTypes.**length** > 0){  
 System.***out***.print(**"throws "**);  
 **for**(**int** i = 0;i < exceptionTypes.**length**;i++){  
 **if**(i == exceptionTypes.**length** - 1){  
 System.***out***.print(exceptionTypes[i].getName());  
 **break**;  
 }  
 System.***out***.print(exceptionTypes[i].getName() + **","**);  
 }  
 }  
 System.***out***.println();  
}

### 获取构造器结构

Class clazz = Person.**class**;  
*//getConstructors():获取当前运行时类中声明为public的构造器*Constructor[] constructors = clazz.getConstructors();  
**for**(Constructor c : constructors){  
 System.***out***.println(c); *//public com.atguigu.java1.Person()*  
}  
  
System.***out***.println();  
*//getDeclaredConstructors():获取当前运行时类中声明的所有的构造器*Constructor[] declaredConstructors = clazz.getDeclaredConstructors();  
**for**(Constructor c : declaredConstructors){  
 System.***out***.println(c);  
}

### 获取父类及父类的泛型

Class clazz = Person.**class**;  
  
Class superclass = clazz.getSuperclass();  
System.***out***.println(superclass); *//class com.atguigu.java1.Creature  
  
//获取运行时类的带泛型的父类*Type genericSuperclass = clazz.getGenericSuperclass();  
System.***out***.println(genericSuperclass); *//com.atguigu.java1.Creature<java.lang.String>*

*获取运行时类的带泛型的父类的泛型*

Class clazz1 = Person.**class**;  
  
Type genericSuperclass = clazz1.getGenericSuperclass();  
ParameterizedType paramType = (ParameterizedType) genericSuperclass;  
*//获取泛型类型*Type[] actualTypeArguments = paramType.getActualTypeArguments();  
System.***out***.println(actualTypeArguments[0].getTypeName()); *//java.lang.String*System.***out***.println(((Class)actualTypeArguments[0]).getName()); *//java.lang.String*

### 6、获取运行时类的接口、所在包、注解等

Class clazz = Person.**class**;  
  
*//获取运行时类实现的接口*Class[] interfaces = clazz.getInterfaces();  
**for**(Class c : interfaces){  
 System.***out***.println(c);  
 *//interface java.lang.Comparable  
 //interface com.atguigu.java1.MyInterface*}  
System.***out***.println();

*//获取运行时类的父类实现的接口*Class[] interfaces1 = clazz.getSuperclass().getInterfaces();  
**for**(Class c : interfaces1){  
 System.***out***.println(c);*//interface java.io.Serializable*}

*//获取运行时类所在的包*Package pack = clazz.getPackage();  
System.***out***.println(pack); *//package com.atguigu.java1*

*//获取运行时类声明的注解*Annotation[] annotations = clazz.getAnnotations();  
**for**(Annotation annos : annotations){  
 System.***out***.println(annos); *//@com.atguigu.java1.MyAnnotation(value=hi)*}

## 调用运行时类的指定结构

### 调用指定属性

在反射机制中，可以直接通过Field类操作类中的属性，通过Field类提供的set()和get()方法就可以完成设置和取得属性内容的操作。

public Field getField(String name) 返回此Class对象表示的类或接口的指定的public的Field。 

public Field getDeclaredField(String name)返回此Class对象表示的类或接口的指定的Field。 

在Field中：

public Object get(Object obj)：取得指定对象obj上此Field的属性内容

public void set(Object obj,Object value)：设置指定对象obj上此Field的属性内容

*//创建运行时类的对象*

Class clazz = Person.class;Person p = (Person) clazz.newInstance();  
  
*//1.getDeclaredField(String fieldName):获取运行时类中指定变量名的属性*Field name = clazz.getDeclaredField(**"name"**);  
*//2.保证当前属性是可访问的*name.setAccessible(**true**);  
*//3.设置指定对象的此属性值*name.set(p,**"Tom"**);  
  
System.***out***.println(name.get(p)); *//Tom*

### 调用指定方法

通过反射，调用类中的方法，通过Method类完成。步骤：

1）通过Class类的getMethod(String name,Class…parameterTypes)方法取得一个Method对象，并设置此方法操作时所需要的参数类型。

2）之后使用Object invoke(Object obj, Object[] args)进行调用，并向方法中传递要设置的obj对象的参数信息。



Object invoke(Object obj, Object … args)

说明：

1.Object 对应原方法的返回值，若原方法无返回值，此时返回null

2.若原方法若为静态方法，此时形参Object obj可为null

3.若原方法形参列表为空，则Object[] args为null

4.若原方法声明为private,则需要在调用此invoke()方法前，显式调用方法对象的

setAccessible(true)方法，将可访问private的方法

*//创建运行时类的对象*Class clazz = Person.**class**;  
Person p = (Person) clazz.newInstance();  
*/\*  
1.获取指定的某个方法  
getDeclaredMethod():参数1：指明获取的方法的名称 参数2：指明获取的方法的形参列表  
 \*/*Method show = clazz.getDeclaredMethod(**"show"**, String.**class**);  
*//2.保证当前方法是可访问的*show.setAccessible(**true**);  
*/\*  
3.调用方法的invoke():参数1：方法的调用者 参数2：给方法形参赋值的实参  
invoke()的返回值即为对应类中调用的方法的返回值。  
 \*/*Object returnValue = show.invoke(p,**"CHN"**); *//String nation = p.show("CHN");*System.***out***.println(returnValue);  
  
System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*如何调用静态方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
  
*//private static void showDesc()*Method showDesc = clazz.getDeclaredMethod(**"showDesc"**);  
showDesc.setAccessible(**true**);  
*//如果调用的运行时类中的方法没有返回值，则此invoke()返回null  
//Object returnVal = showDesc.invoke(null);*Object returnVal = showDesc.invoke(Person.**class**);  
System.***out***.println(returnVal); *//null*

### 调用指定构造器（了解）

Class clazz = Person.**class**;  
  
*//private Person(String name)  
//获取指定的构造器：getDeclaredConstructor():参数：指明构造器的参数列表*Constructor constructor = clazz.getDeclaredConstructor(String.**class**);  
  
*//2.保证此构造器是可访问的*constructor.setAccessible(**true**);  
  
*//3.调用此构造器创建运行时类的对象*Person per = (Person) constructor.newInstance(**"Tom"**);  
System.***out***.println(per);

## 反射的应用：动态代理

### 动态代理概设

1. 代理设计模式的原理

使用一个代理将对象包装起来, 然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。 

1. 静态代理

特征是代理类和目标对象的类都是**在编译期间确定下来**，不利于程序的扩展。同时，每一个代理类只能为一个接口服务，这样一来程序开发中必然产生过多的代理。最好可以通过一个代理类完成全部的代理功能。

*//代理类***class** ProxyClothFactory **implements** ClothFactory{  
  
 *//用被代理类对象进行实例化* **private** ClothFactory **factory**;

**public** ProxyClothFactory(ClothFactory factory){  
 **this**.**factory** = factory;  
 }  
 @Override  
 **public void** produceCloth() {  
 System.***out***.println(**"代理工厂做一些准备工作"**);  
 **factory**.produceCloth();  
 System.***out***.println(**"代理工厂做一些后续的收尾工作"**);  
 }  
}  
*//被代理类（在编译期间确定下来，不具有动态性）***class** NikeClothFactory **implements** ClothFactory{  
  
 @Override  
 **public void** produceCloth() {  
 System.***out***.println(**"Nike工厂生产一批运动服"**);  
 }  
}  
**public class** StaticProxyTest {  
 **public static void** main(String[] args) {  
 *//创建被代理类的对象* ClothFactory nike = **new** NikeClothFactory(); *//多态的体现*  
 *//创建代理类的对象* ClothFactory proxyClothFactory = **new** ProxyClothFactory(nike);  
 *//调用被代理类的方法*  
 proxyClothFactory.produceCloth();  
 }  
}

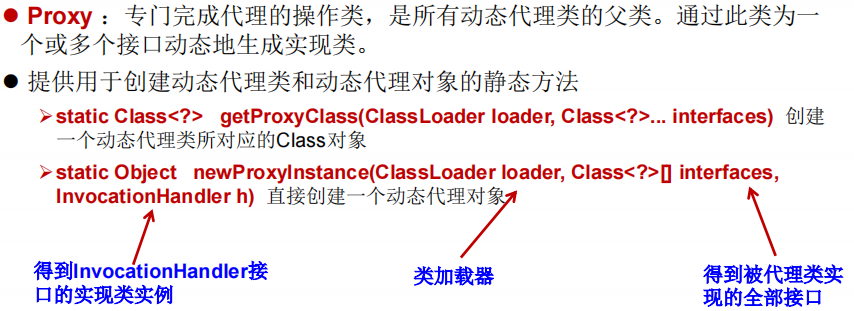
1. 动态代理

是指客户通过代理类来调用其它对象的方法，并且是在**程序运行时根据需要动态创建**目标类的代理对象。动态代理使用场合: 调试、远程方法调用

4）动态代理相比于静态代理的优点

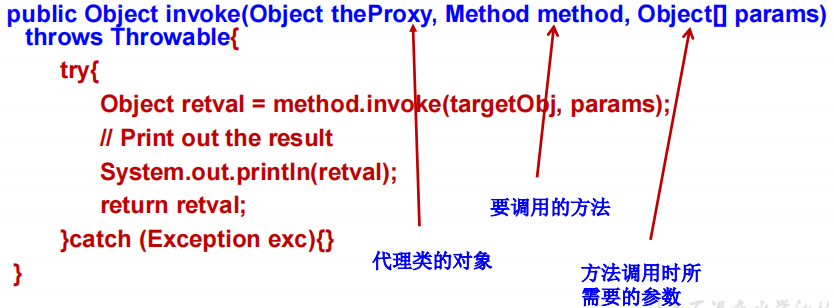
抽象角色中（接口）声明的所有方法都被转移到调用处理器一个集中的方法中处理，这样，我们可以更加灵活和统一的处理众多的方法。

### Java动态代理相关API

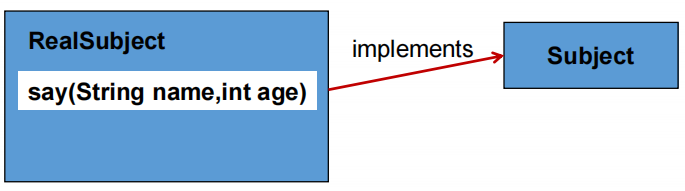


### 动态代理的步骤

1）创建一个实现接口InvocationHandler的类，它必须实现invoke方法，以完成代理的具体操作。



2）创建被代理的类以及接口



1. 通过Proxy的静态方法newProxyInstance(ClassLoader loader, Class[] interfaces,

InvocationHandler h) 创建一个接口代理

*//调用此方法，返回一个代理类的对象。***public static** Object getProxyInstance(Object obj){*//obj:被代理类的对象* MyInvocationHandler handler = **new** MyInvocationHandler();  
  
 handler.bind(obj);  
  
 *//创建代理类的对象*

*Proxy.newProxyInstance(类的加载器、类的接口、调用被代理类里面的方法(handler是桥梁的作用))* **return** Proxy.*newProxyInstance*(obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),handler);  
}

5）通过代理类的对象，调用被代理类的方法

*//声明接口*

**interface** Human{  
 String getBelief();  
 **void** eat(String food);  
}  
*//1、被代理类实现接口***class** SuperMan **implements** Human{  
  
 @Override  
 **public** String getBelief() {  
 **return "I believe I can fly!"**;  
 }  
 @Override  
 **public void** eat(String food) {  
 System.***out***.println(**"我喜欢吃"** + food);  
 }  
}  
*/\*  
要想实现动态代理，需要解决的问题？  
问题一：如何根据加载到内存中的被代理类，动态的创建一个代理类及其对象。  
问题二：当通过代理类的对象调用方法a时，如何动态的去调用被代理类中的同名方法a。  
 \*/  
//2、动态创建代理类的对象***class** ProxyFactory{  
 *//调用此方法，返回一个代理类的对象。解决问题一* **public static** Object getProxyInstance(Object obj){*//obj:被代理类的对象* MyInvocationHandler handler = **new** MyInvocationHandler();  
  
 handler.bind(obj);  
  
 *//创建代理类的对象：Proxy.newProxyInstance(类的加载器、类的接口、调用被代理类里面的方法(handler是桥梁的作用))* **return** Proxy.*newProxyInstance*(obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),handler);  
 }  
}  
*//3、动态的去调用被代理类中的同名方法*  
**class** MyInvocationHandler **implements** InvocationHandler{  
  
 **private** Object **obj**;*//需要使用被代理类的对象进行赋值* **public void** bind(Object obj){  
 **this**.**obj** = obj;  
 }  
 *//当我们通过代理类的对象，调用方法a时，就会自动的调用如下的方法：invoke()  
 //将被代理类要执行的方法a的功能就声明在invoke()中* @Override  
 **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  
 *//method:即为代理类对象调用的方法，此方法也就作为了被代理类对象要调用的方法  
 //obj:被代理类的对象* Object returnValue = method.invoke(**obj**,args);  
  
 *//上述方法的返回值就作为当前类中的invoke()的返回值。* **return** returnValue;  
 }  
}

*//测试*  
**public class** ProxyTest {  
  
 **public static void** main(String[] args) {  
 *//1、创建被代理类对象* SuperMan superMan = **new** SuperMan();  
 *//2、proxyInstance:创建代理类的对象* Human proxyInstance = (Human) ProxyFactory.*getProxyInstance*(superMan);  
 *//3、当通过代理类对象调用方法时，会自动的调用被代理类中同名的方法*  
 System.***out***.println(proxyInstance.getBelief());  
 proxyInstance.eat(**"四川麻辣烫"**);

System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);

*//1、创建被代理类对象（动态创建被代理类的对象，体现反射机制）* NikeClothFactory nikeClothFactory = **new** NikeClothFactory();  
 *//2、创建代理类对象* ClothFactory proxyClothFactory = (ClothFactory) ProxyFactory.*getProxyInstance*(nikeClothFactory);  
 *//3、通过代理类的对象，调用被代理类的方法* proxyClothFactory.produceCloth();  
 }  
}

*输出：I believe I can fly!*

*我喜欢吃四川麻辣烫*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Nike工厂生产一批运动服*

### 动态代理与AOP

# Java8的其它新特性

## Lambda表达式

Lambda是一个匿名函数，我们可以把Lambda表达式理解为是一段可以传递的代码（将代码像数据一样进行传递）

### Lambda表达式的使用

1）举例： (o1,o2) -> Integer.compare(o1,o2);

2）格式：

-> :lambda操作符 或 箭头操作符

->左边：lambda形参列表 （其实就是接口中的抽象方法的形参列表）

->右边：lambda体 （其实就是重写的抽象方法的方法体）

3）Lambda表达式的本质：作为函数式接口的实例

4）如果一个接口中，只声明了一个抽象方法，则此接口就称为函数式接口。我们可以在一个接口上使用 @FunctionalInterface 注解，这样做可以检查它是否是一个函数式接口。

5）所以以前用匿名实现类表示的现在都可以用Lambda表达式来写。

### Lambda表达式的6种语法

语法格式一：无参，无返回值

Runnable r1 = **new** Runnable() {  
 @Override  
 **public void** run() {  
 System.***out***.println(**"我爱北京天安门"**); *//我爱北京天安门* }  
};  
r1.run();  
  
Runnable r2 = () -> System.***out***.println(**"我爱北京故宫"**); *//我爱北京故宫*r2.run();

语法格式二：一个参数，无返回值

Consumer<String> con = **new** Consumer<String>() {  
 @Override  
 **public void** accept(String s) {  
 System.***out***.println(s);  
 }  
};  
con.accept(**"谎言和誓言的区别是什么？"**); *//谎言和誓言的区别是什么？*  
  
Consumer<String> con1 = (String s) -> System.***out***.println(s);  
con1.accept(**"都当真了"**); *//都当真了*

语法格式三：类型推断

数据类型可以省略，因为可由编译器推断得出，称为“类型推断”

Consumer<String> con1 = (String s) -> {  
 System.***out***.println(s);  
};  
con1.accept(**"当真了"**);   
  
Consumer<String> con2 = (s) -> System.***out***.println(s);  
con2.accept(**"当真了"**);

语法格式四：只有一个参数

Lambda 若只需要一个参数时，参数的小括号可以省略

Consumer<String> con1 = (s) -> System.***out***.println(s);  
con1.accept(**"当真了"**);  
  
Consumer<String> con2 = s -> System.***out***.println(s);  
con2.accept(**"当真了"**);

语法格式五：Lambda有多个参数

Lambda需要两个或以上的参数，多条执行语句，并且可以有返回值

Comparator<Integer> com1 = **new** Comparator<Integer>() {  
 @Override  
 **public int** compare(Integer o1, Integer o2) {  
 System.***out***.println(o1);  
 System.***out***.println(o2);  
 **return** o1.compareTo(o2);  
 }  
};  
System.***out***.println(com1.compare(12,21));  
  
Comparator<Integer> com2 = (o1,o2) -> {  
 System.***out***.println(o1);  
 System.***out***.println(o2);  
 **return** o1.compareTo(o2);  
};  
System.***out***.println(com2.compare(12,6));

语法格式六：Lambda体只有一条语句

当Lambda体只有一条语句时，return与大括号若有，都可以省略

Comparator<Integer> com1 = (o1,o2) -> {  
 **return** o1.compareTo(o2);  
};  
System.***out***.println(com1.compare(12,6));

Comparator<Integer> com2 = (o1,o2) -> o1.compareTo(o2);  
System.***out***.println(com2.compare(12,21));

语法总结：

->左边：lambda形参列表的参数类型可以省略(类型推断)；如果lambda形参列表只有一个参数，其一对()也可以省略

->右边：lambda体应该使用一对{}包裹；如果lambda体只有一条执行语句（可能是return语句），省略这一对{}和return关键字

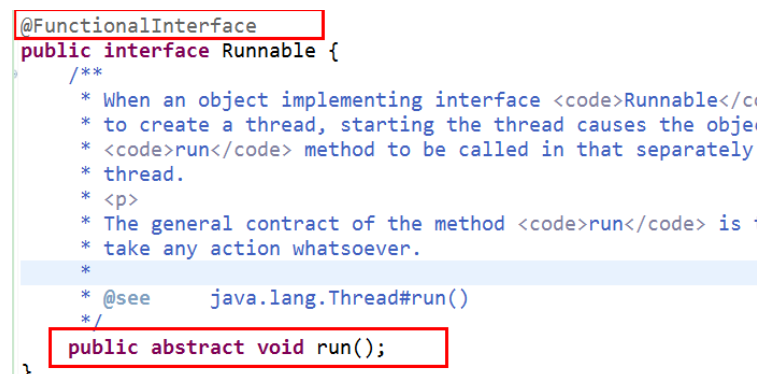
## 函数式(Functional)接口

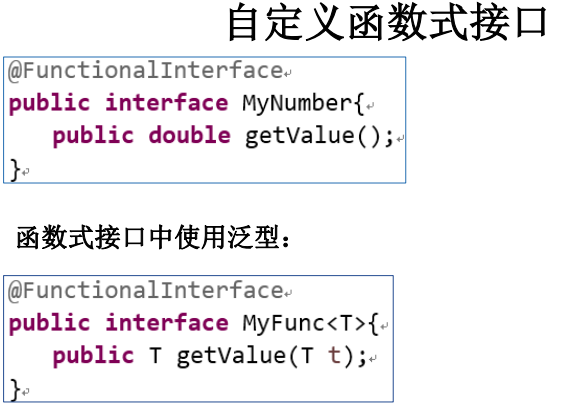
### 函数式接口概述

1）函数式接口：只包含一个抽象方法的接口。可以通过Lambda表达式来创建该接口的对象。只要一个对象是函数式接口的实例，那么该对象就可以用Lambda表达式来表示。

2）可以在一个接口上使用@FunctionalInterface注解，这样做可以检查它是否是一个函数式接口。同时javado也会包含一条声明，说明这个接口是一个函数式接口。

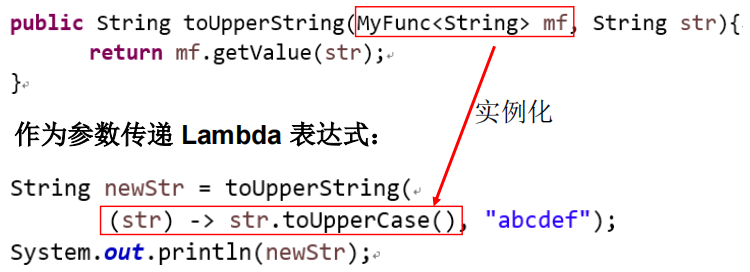
3）在java.util.function包下定义Java 8的丰富的函数式接口





### 作为参数传递Lambda表达式

为了将Lambda表达式作为参数传递，接Lambda表达式的参数类型必须是与该Lambda表达式兼容的函数式接口的类型。



### Java 内置四大核心函数式接口



1）消费型接口 Consumer<T> void accept(T t)

**public void** test1(){

*//方式一：*  
 happyTime(500, **new** Consumer<Double>() {  
 @Override  
 **public void** accept(Double aDouble) {  
 System.***out***.println(**"价格为："** + aDouble);  
 }  
 });  
 *//方式二：*  
 happyTime(400,money -> System.***out***.println(**"价格为："** + money));  
}  
**public void** happyTime(**double** money, Consumer<Double> con){  
 con.accept(money);  
}

2）断定型接口 Predicate<T> boolean test(T t)

**public void** test2(){  
 List<String> list = Arrays.*asList*(**"北京"**,**"南京"**,**"天津"**,**"东京"**,**"西京"**,**"普京"**);  
 *//方式一：*  
 List<String> filterStrs = filterString(list, **new** Predicate<String>() {  
 @Override  
 **public boolean** test(String s) {  
 **return** s.contains(**"京"**);  
 }  
 });

*//方式二：*  
 List<String> filterStrs1 = filterString(list,s -> s.contains(**"京"**));  
}  
  
*//根据给定的规则，过滤集合中的字符串。此规则由Predicate的方法决定***public** List<String> filterString(List<String> list, Predicate<String> pre){  
  
 ArrayList<String> filterList = **new** ArrayList<>();  
 **for**(String s : list){  
 **if**(pre.test(s)){  
 filterList.add(s);  
 }  
 }  
 **return** filterList;  
}

## 方法引用与构造器引用

### 方法引用概述

当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用。

方法引用可以看做是Lambda表达式深层次的表达。换句话说，方法引用就是Lambda表达式，也就是函数式接口的一个实例，通过方法的名字来指向一个方法，可以认为是Lambda表达式的一个语法糖。

要求：实现接口的抽象方法的参数列表和返回值类型，必须与方法引用的方法的参数列表和返回值类型保持一致！

格式：使用操作符 “::” 将类(或对象) 与 方法名分隔开来。

如下三种主要使用情况： 

对象::实例方法名

 类::静态方法名

 类::实例方法名

### 方法引用的三种使用情况

方法引用使用的要求：要求接口中的抽象方法的形参列表和返回值类型与方法引用的方法的形参列表和返回值类型相同！（针对于情况1和情况2）

情况一：对象 :: 实例方法（非静态方法）

*//Consumer中的void accept(T t)  
//PrintStream中的void println(T t)*  
**public void** test1() {  
 Consumer<String> con1 = str -> System.***out***.println(str);  
 con1.accept(**"北京"**);  
  
 Consumer<String> con2 = System.***out***::println;  
 con2.accept(**"beijing"**);  
}  
  
*//Supplier中的T get()  
//Employee中的String getName()*  
**public void** test2() {  
 Employee emp = **new** Employee(1001,**"Tom"**,23,5600);  
  
 Supplier<String> sup1 = () -> emp.getName();

System.out.println(sup1.get());  
  
 Supplier<String> sup2 = emp::getName;

System.out.println(sup2.get());  
}

情况二：类 :: 静态方法

*//Comparator中的int compare(T t1,T t2)  
//Integer中的int compare(T t1,T t2)*  
**public void** test3() {  
 Comparator<Integer> com1 = (t1,t2) -> Integer.*compare*(t1,t2);  
 System.***out***.println(com1.compare(12,21));  
  
 Comparator<Integer> com2 = Integer::*compare*;  
 System.***out***.println(com2.compare(12,3));  
}  
  
*//Function中的R apply(T t)  
//Math中的Long round(Double d)*  
**public void** test4() {  
 Function<Double,Long> func = **new** Function<Double, Long>() {  
 @Override  
 **public** Long apply(Double d) {  
 **return** Math.*round*(d);  
 }  
 };  
 Function<Double,Long> func1 = d -> Math.*round*(d);  
 System.***out***.println(func1.apply(12.3));  
  
 Function<Double,Long> func2 = Math::*round*;  
 System.***out***.println(func2.apply(12.6));  
}

情况三：类 :: 实例方法（非静态方法）

*// Comparator中的int comapre(T t1,T t2)  
// String中的int t1.compareTo(t2)*  
**public void** test5() {  
 Comparator<String> com1 = (s1,s2) -> s1.compareTo(s2);  
 System.***out***.println(com1.compare(**"abc"**,**"abd"**));  
  
 Comparator<String> com2 = String :: compareTo;  
 System.***out***.println(com2.compare(**"abd"**,**"abm"**));  
}  
  
*//BiPredicate中的boolean test(T t1, T t2);  
//String中的boolean t1.equals(t2)*  
**public void** test6() {  
 BiPredicate<String,String> pre1 = (s1,s2) -> s1.equals(s2);  
 System.***out***.println(pre1.test(**"abc"**,**"abc"**));  
  
 BiPredicate<String,String> pre2 = String :: equals;  
 System.***out***.println(pre2.test(**"abc"**,**"abd"**));  
}  
  
*//Function中的R apply(T t)  
//Employee中的String getName();*  
**public void** test7() {  
 Employee employee = **new** Employee(1001, **"Jerry"**, 23, 6000);  
  
 Function<Employee,String> func1 = e -> e.getName();  
 System.***out***.println(func1.apply(employee));  
  
 Function<Employee,String> func2 = Employee::getName;  
 System.***out***.println(func2.apply(employee));  
}

### 构造器引用

### 数组引用

## 强大的Stream API

### Stream API概述

1）Stream是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。“集合讲的是数据，Stream讲的是计算！”

2）Stream和Collection集合的区别：Collection是一种静态的内存数据结构，而Stream是有关计算的。前者是主要面向内存，存储在内存中，后者主要是面向CPU，通过CPU实现计算

注意：

① Stream自己不会存储元素，数据仍在集合中

② Stream不会改变源对象。相反，他们会返回一个持有结果的新Stream。

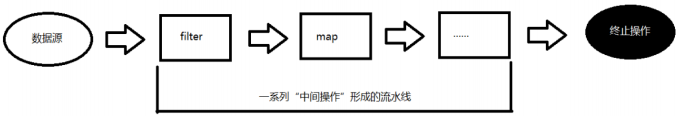
③ Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

3）Stream 的操作三个步骤

1- 创建Stream：一个数据源（如：集合、数组），获取一个流

2- 中间操作：一个中间操作链，对数据源的数据进行处理

3- 终止操作(终端操作) 一旦执行终止操作，就执行中间操作链，并产生结果。之后不会再被使用



### 创建Stream的四种方式



方式一：通过集合

List<Employee> employees = EmployeeData.*getEmployees*();  
  
*//返回一个顺序流*Stream<Employee> stream = employees.stream();  
*//返回一个并行流*Stream<Employee> parallelStream = employees.parallelStream();

方式二：通过数组

**int**[] arr = **new int**[]{1,2,3,4,5,6};  
*//调用Arrays类的static <T> Stream<T> stream(T[] array): 返回一个流*IntStream stream = Arrays.*stream*(arr);  
  
Employee e1 = **new** Employee(1001,**"Tom"**);  
Employee e2 = **new** Employee(1002,**"Jerry"**);  
Employee[] arr1 = **new** Employee[]{e1,e2};  
Stream<Employee> stream1 = Arrays.*stream*(arr1);

方式三：通过Stream的of()

Stream<Integer> stream = Stream.*of*(1, 2, 3, 4, 5, 6);

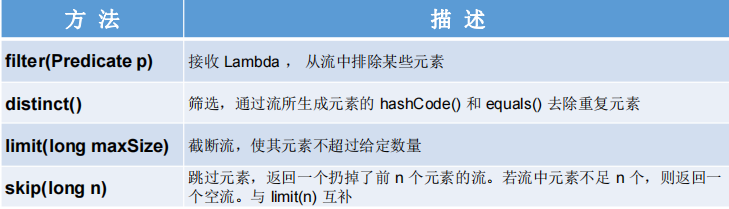
方式四：创建无限流（了解）

*迭代  
//public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)  
//遍历前10个偶数*Stream.*iterate*(0, t -> t + 2).limit(10).forEach(System.***out***::println);  
  
*//生成  
//public static<T> Stream<T> generate(Supplier<T> s)*Stream.*generate*(Math::*random*).limit(10).forEach(System.***out***::println);

### Stream的中间操作

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何的处理！而在终止操作时一次性全部处理，称为“惰性求值”。

#### 筛选与切片

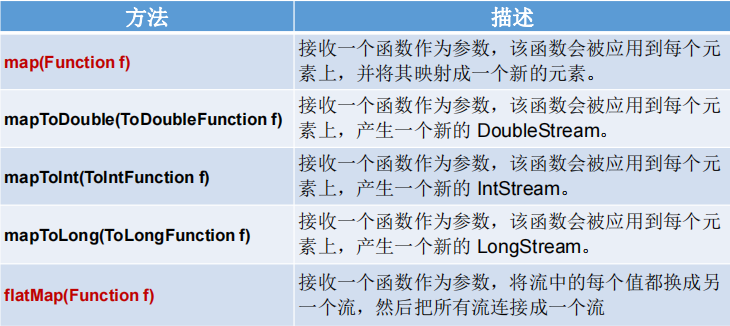


List<Employee> list = EmployeeData.*getEmployees*();Stream<Employee> stream = list.stream();  
*//1、查询员工表中薪资大于7000的员工信息*stream.**filter**(e -> e.getSalary() > 7000).forEach(System.***out***::println);

*//2、查询前三条员工信息*list.stream().**limit(3)**.forEach(System.***out***::println);  
  
*//3、跳过前三条数据，查询后面的员工信息*list.stream().**skip(3)**.forEach(System.***out***::println);  
  
*//3、跳过前三条数据，查询后面的员工信息*list.add(**new** Employee(1010,**"刘强东"**,40,8000));  
list.add(**new** Employee(1010,**"刘强东"**,41,8000));

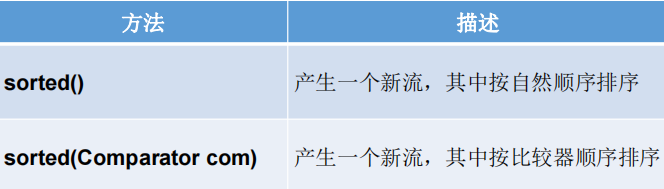
*//4、数据按照hashCode() 和 equals()去重*list.stream().**distinct()**.forEach(System.***out***::println);

#### 映射



**public void** test2(){List<String> list = Arrays.*asList*(**"aa"**, **"bb"**, **"cc"**, **"dd"**);  
 *//1、将list的元素全部大写后返回* list.stream().**map**(str -> str.toUpperCase()).forEach(System.***out***::println);  
  
 *//2、获取员工姓名长度大于3的员工的姓名。* List<Employee> employees = EmployeeData.*getEmployees*();  
 Stream<String> namesStream = employees.stream().map(Employee::getName);  
 namesStream.filter(name -> name.length() > 3).forEach(System.***out***::println);  
 System.***out***.println();  
 *//3、* Stream<Stream<Character>> streamStream = list.stream().map(StreamAPITest1::*fromStringToStream*);  
 streamStream.forEach(s ->{  
 s.forEach(System.***out***::println);  
 });  
 System.***out***.println();  
*// flatMap(Function f)——接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。* Stream<Character> characterStream = list.stream().flatMap(StreamAPITest1::*fromStringToStream*);  
 characterStream.forEach(System.***out***::println);  
}  
*//将字符串中的多个字符构成的集合转换为对应的Stream的实例***public static** Stream<Character> fromStringToStream(String str){*//aa* ArrayList<Character> list = **new** ArrayList<>();  
 **for**(Character c : str.toCharArray()){  
 list.add(c);  
 }  
 **return** list.stream();  
}

#### 排序



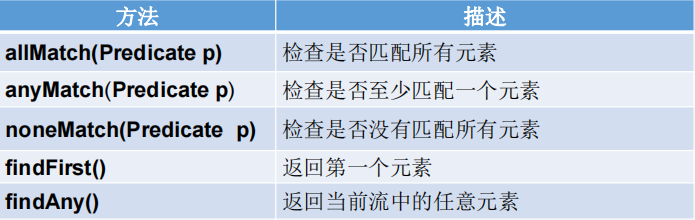
*//sorted()——自然排序*List<Integer> list = Arrays.*asList*(12, 43, 65, 34, 87, 0, -98, 7);  
list.stream().sorted().forEach(System.***out***::println);

*//抛异常，原因:Employee没有实现Comparable接口  
//List<Employee> employees = EmployeeData.getEmployees();  
//employees.stream().sorted().forEach(System.out::println);  
  
//2、sorted(Comparator com)——定制排序*List<Employee> employees = EmployeeData.*getEmployees*();  
employees.stream().sorted( (e1,e2) -> {  
 **int** ageValue = Integer.*compare*(e1.getAge(),e2.getAge());  
 **if**(ageValue != 0){  
 **return** ageValue;  
 }**else**{  
 **return** -Double.*compare*(e1.getSalary(),e2.getSalary());  
 }  
}).forEach(System.***out***::println);

### Stream的终止操作

* 终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，如：List、Integer，甚至是void
* 流进行了终止操作后，不能再次使用

#### 匹配与查找

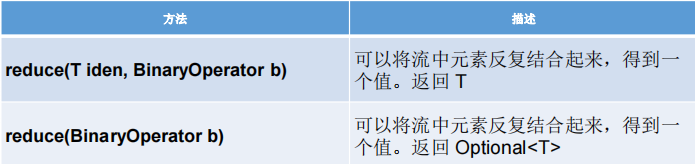


List<Employee> employees = EmployeeData.*getEmployees*();  
 *//1、是否所有的员工的年龄都大于18***boolean** allMatch = employees.stream().**allMatch**(e -> e.getAge() > 18);  
  
*//2、是否存在员工的工资大于10000***boolean** anyMatch = employees.stream().**anyMatch**(e -> e.getSalary() > 10000);  
  
*//3、是否存在员工姓“雷”***boolean** noneMatch = employees.stream().**noneMatch**(e -> e.getName().startsWith(**"雷"**));  
  
*//4、返回第一个元素*Optional<Employee> employee = employees.stream().**findFirst**();  
  
*//5、返回当前流中的任意元素*Optional<Employee> employee1 = employees.parallelStream().**findAny**();



List<Employee> employees = EmployeeData.*getEmployees*();  
*//1、count——返回流中元素的总个数***long** count = employees.stream().filter(e -> e.getSalary() > 5000).**count**();  
  
*//2、返回最高工资*Stream<Double> salaryStream = employees.stream().map(e -> e.getSalary());  
Optional<Double> maxSalary = salaryStream.**max**(Double::*compare*);  
  
*//3、返回最低工资的员工*Optional<Employee> employee = employees.stream().**min**((e1, e2) -> Double.*compare*(e1.getSalary(), e2.getSalary()));  
  
*//4、forEach——内部迭代*employees.stream().**forEach**(System.***out***::println);  
  
*//5、使用集合的遍历操作*employees.**forEach**(System.***out***::println);

#### 归约



*//练习1：计算1-10的自然数的和*List<Integer> list = Arrays.*asList*(1,2,3,4,5,6,7,8,9,10);  
Integer sum = list.stream().reduce(0, Integer::*sum*);

*//练习2：计算公司所有员工工资的总和*List<Employee> employees = EmployeeData.*getEmployees*();  
Stream<Double> salaryStream = employees.stream().map(Employee::getSalary);  
*//Optional<Double> sumMoney = salaryStream.reduce(Double::sum);*Optional<Double> sumMoney = salaryStream.reduce((d1,d2) -> d1 + d2);  
System.***out***.println(sumMoney.get());

#### 收集



*练习1：查找工资大于6000的员工，结果返回为一个List或Set*List<Employee> employees = EmployeeData.*getEmployees*();  
List<Employee> employeeList = employees.stream().filter(e -> e.getSalary() > 6000).**collect(Collectors.*toList*())**;  
employeeList.forEach(System.***out***::println);  
  
Set<Employee> employeeSet = employees.stream().filter(e -> e.getSalary() > 6000).**collect(Collectors.toSet())**;  
employeeSet.forEach(System.***out***::println);

## Optional类