|  |
| --- |
|  |

|  |
| --- |
| **久 杨 珍 精** |
|  |
|  |
| 悟尽归原，几近于道。 |
|  |
| create by Yangcl |
| 2016/2/3 |
|  |

1 转换与运算符

* 1. 进制转换

1.1.1 十进制和二进制

* Ox99836 十进制转二进制

原理：对十进制进行除2运算，然后一直找余数，如果能够整除则记录为0，否则1，整个过程从右向左累加0或1。

下面计算6和9的二进制表现形式：

|  |  |  |
| --- | --- | --- |
| 除数 | 被除数 | 余数累加 |
| 整数6的二进制运算原理 | | |
| 2 | 6 | 0 |
| 2 | 3 | 10 |
| 2 | 1 | 110 |
| 所以6的二进制等于 110 | | |
|  | | |
| 整数9的二进制运算原理 | | |
| 2 | 9 | 1 |
| 2 | 4 | 01 |
| 2 | 2 | 001 |
| 2 | 1 | 1001 |
| 所以9的二进制等于 1001 | | |

9÷2=4余数为1，不能被整除，所以第一次记录为1，累加=1

4÷2=2余数为0，能被整除，所以第二次记录为0，累加=01

2÷2=1余数为0，能被整除，所以第三次记录为0，累加=001

1÷2 不再为整数，余数为1，不能被整除，所以第四次记录为1，累加=1001

* Ox99837 二进制转十进制

原理：二进制的0或1乘以2的n次方的过程。下表演示1110101的转换。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 二进制 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 2的n次方 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
| 计算 | 1\*64 | 1\*32 | 1\*16 | 0\*8 | 1\*4 | 0\*2 | 1\*1 |
| 0乘以任何数都等于0，1乘以任何数还等于那个数；  故简化如下： | | | | | | | |
| 简化 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 求和 | 64 + 32 + 16 + 4 + 1 = 117 | | | | | | |

【表格1.1-1】

111 1111 的十进制是127

1111 1111 的十进制是127 + 128 = 255；所以一个字节的最大值是255。

* Ox99838 二进制加法运算

有了【表格1.1-1】的对照，讨论二进制加法就更方便了。下面举例 7 + 3 = 10的例子，二进制加法与十进制加法原理相同。

|  |  |  |
| --- | --- | --- |
| + | 111 | 1 + 2 + 4= 7 |
| 011 | 1 + 2 = 3 |
| = | 1010 |  |
| 1010对应【表格1.1-1】的8+2 = 10 | | |

这就是在内存中的计算。

1.1.2 十六进制和八进制

* Ox99851二进制与十六进制

十六进制包括：0123456789ABCDE；共15个元素。之所以是15个元素是因为逢16进一，和十进制没有10是一样的。针对二进制与十六进制之间的转换这里用【表格1.1-1】中的数据“111-0101” 来举例，它的十进制是117。

在十六进制的表示规则中，四个二进制位(如：01则表示2个二进制位)就是一个十六进制位，如果二进制串不满足4的整数倍则在其左侧进行补零来计算。但问题来了，为什么要用4个二进制位？因为“0000”= 0，“1111”= 15，15是最大值，满足16则进位。这就是十六进制的由来。同理可知，八进制采用3个二进制位，因为“111”= 7，逢8则进一。

计算规则示例如下：

|  |  |
| --- | --- |
| 补零： | 111-0101 |
| 0111-0101 |
| 0x： | 7 5 |
| 所以117的十六进制为0x75. | |

再举个例子：“1001-1010”= 154，它的十六进制为0x9A。

* Ox99852二进制与八进制

|  |  |
| --- | --- |
| 补零： | 111-0101 |
| 001-110-101 |
|  | 1 6 5 |
| 所以117的八进制为165. | |

1.2 类型转换与精度丢失

02-08-类型转换

1.3 转义字符

1.4 位运算符

1.4.1 左移|右移

1.4.2 与|或|异或

异或与加密原理

2 面向对象

这一章非常重要，对Java面向对象理解的深浅将决定你是一名设计者还是一名码农。

如果你无法 **融会贯通** 这一章的思想，事实上你并不会Java。Java入门容易，精通难正因如此。本章取用了毕向东老师的主体思想，以及网络上几位资深Java一线开发者博客中的经验；经过与Java斗士猛男涛的反复推敲，现将部分个人见解整理如下，受个人阅历与实际工作经验所限，必有大量不足、欠妥之处，希望各位高手指出，如果正确合理，我会立即修改，并附上您的署名。

——致程序员的情怀。

2.1类与对象的关系

2.1.1 类、对象、实例 和 static成员

在Java核心技术与Java编程思想中，对【对象】的定义是：万物皆对象。可见，对象是一个包含万物的全集，所以类是对象的子集，实例也是对象的子集。有人说“对象即实例”，就显得有些空泛，对象包含的内容多了，可未必就是实例。但是，实例即对象一定是正确的。

再来看【类】这个概念。Java核心技术对其定义如下：类是构造对象的模板或蓝图。

简单清晰，类就是一个**模板**。可以将“类”看成做棒棒糖的模子，而每一个你拿在手里的棒棒糖就是棒棒糖类的【实例】。

由【类】构造(construst)对象的过程称为创建类的【实例】(instance)。这个构造对象的过程就是你调用这个类的**构造函数**的语句，如：**new** Car(); 这句代码就创建了一个Car类的实例，这个实例会被分配到Java的堆内存中。

**public** **class** Car {

**private** String color = "黑色";

**private** **int** door = 4;

**private** **static** **int** *num* = 4;

**public** **void** run(){

System.***out***.println(color + " 车有" + door + "个门，开动了");

}

}

}

【代码2.1-1】

【类】是对一种具体事务的属性定义，比如说“人”这个类，就是对人的一切属性的定义，包括肤色、语言、人种、模样、是否残疾、2个眼睛、一个鼻子和需要食物、空气来“运行”等等。你自己可以说你是一个人，但你绝不是人这个类；因为你是人类定义范畴内的一个具体的、现实中的“真实人”个体，包含了人类定义范畴内的所有属性与行为。

属性对应类中的变量，即Filed(字段如：private String name;)；行为对应类中的函数(方法)。当你定义一个类的时候，就是在对这一类事物进行抽象层面的描述，就是在表达它所存在的属性和行为。属性和行为都称作这个类中的成员，包括成员变量和成员方法。

【static】在介绍静态的时候需要引入一个概念：域。对应类中的**变量**，即Filed；在一个类中，包含静态变量与非静态变量，非静态变量又称为【实例域】，静态变量又称作【静态域】；实例域由类实例来访问，静态域由类直接访问，但类实例也可以访问并改变其值。

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

**new** Car().color = "红色"; // 实例域由类实例来访问

Car.*num* = 6; // 静态域由类直接访问

**new** Car().*num* = 8; // 但是类实例也可以调用静态域

System.***out***.println(Car.*num*); // = 8

}

**new** Car().*num* = 8; 这种是不提倡的。

实例域属于类实例本身，静态域则属于类本身。在一个类实例对象初始化的过程中，对于这个类中定义的所有实例域都有一份自己的拷贝。比如，你new 1000个Car对象则有1000个实例域color和door，但是只有一个静态域num。即使没有一个Car实例，静态域num也是存在的，因为静态域从属于类本身。故静态域也称为【类域】。

实例域只跟实例对象有关，同一个类的不同类实例有不同的实例域；而类域，跟类有关被这个类的所有类实例所共享。

注意，【static】这一段讨论的都是类中的变量，不是类的方法。

对于“static”这个关键字，感觉Java对它的处理是没地方去放了，索性由他去哪儿都可以让他存在。原引《java核心技术 卷I》原书第八版中110页注释中的描述如下：

“*在绝大多数的面向对象程序设计语言中，静态域被称为类域。术语‘static’只是沿用了C++ 中的叫法，并无实际意义。*”

对于被static所修饰的变量和方法，是无法被Java垃圾回收的，只要你的程序还在运行，这部分资源就会被占用。故在实际开发中使用static会非常谨慎，通常一些工具类会被系统架构者封装成为统一的Java类，省去对象创建的时间，在内存中快速使用；比如StringUtil、HttpClientUtil和Md5Util等等。

2.1.2 类实例的内存地址与堆栈结构关系

即：对象的初始化过程。

* 0x3258b 基本原理

实例域的赋值变化等。

类类型变量指向类实例

表格内的4行代码按照其执行顺序，以数字来标识其堆栈内存变化顺序。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 栈内存 | | 堆内存变化 | | | |
|  | | Car c = new Car(); | | | |
| c.color = "蓝色"; | | | |
| Car d = new Car(); | | | |
| d.run(); | | | |
|  | 堆内存实例变化顺序(从右至左) | | |
| 类 类型变量 |  | **内存地址** | **类变量重新赋值** | **字段默认值** | **堆内存默认初始化值** |
| 1 » Car c  6 » 0x9ab68 | 5 » 0x9ab68 | 4 » color = "蓝色" | 3 » color="黑色" | 2 » color=null |
| 否 door=4 | 3 » door=4 | 2 » door=0 |
| 否 num=4 | 3 » num=4 | 2 » num=0 |
|  |  | | | |
| 1 » Car d  5 » 0x966cd | 4 » 0x966cd | 否 color="黑色" | 3 » color="黑色" | 2 » color=null |
| 否 door=4 | 3 » door=4 | 2 » door=0 |
| 否 num=4 | 3 » num=4 | 2 » num=0 |
|  | | | | | |

* 0x3258c 高级扩展 多引用指向同一实例对象

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 栈内存 | | 堆内存变化 | | | |
|  | | Car c = new Car(); | | | |
| c.color = "蓝色"; | | | |
| Car d = c; | | | |
| d.door = 5; | | | |
| c.run(); | | | |
|  | 堆内存实例变化顺序(从右至左) | | |
| 类 类型变量 |  | **内存地址** | **类变量重新赋值** | **字段默认值** | **堆内存默认初始化值** |
| 1 » Car c  6 » 0x00008 | 5 » 0x00008 | 4 » color = "蓝色" | 3 » color="黑色" | 2 » color=null |
| 否 door=4 | 3 » door=4 | 2 » door=0 |
| 否 num=4 | 3 » num=4 | 2 » num=0 |
|  |  | | | |
| 7 » Car d  8 » 0x00008 |  | color = "蓝色" |  |  |
| 9 » **door=5** |  |  |
| 否 num=4 |  |  |
|  | | | | | |

所以，c.run()打印出来的应该是 蓝色 和 5

2.2构造函数 与 构造代码块

2.2.1 构造函数

一种特殊的函数，其主要功能是用来在创建对象实例时初始化对象。

**public** **class** Person {

**private** String name;

**private** **int** age;

**public** Person() {

}

**public** Person(String name) {

**this**.name = name;

}

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

**this**.name = name;

**this**.age = age;

}

}

一个类的构造函数可以有多个，分别对应初始化**实例域**中不同的字段(注意这里是实例域，不是类域，因为构造函数是非静态的函数，无法调用静态变量)。所谓的初始化就是在给字段赋值，这项工作和getter / setter函数相类似，尤其是在实体类中。

每个类中都有一个默认的构造函数，不需要进行显示的声明，他不初始化任何实例域中的字段，称作：默认构造函数；但当你在类中写了带参数的构造函数后，并且没有显示的去声明默认构造函数，那么默认构造函数则被覆盖。此时如果项目中有别的地方用了这个类的默认构造方法就会报错。所以一般都会像上面的代码，把默认的和自己的都写上。

在实际项目中，自定义构造函数通常用于改造别人的代码，以最小的侵入性来完成新的功能需求。比如项目中有一个定时任务(web quartz job)用于定时拉去价格变动的商品信息

，变动时间范围在10点到12点的，12点到14点的等等，都是程序根据服务器当前时间自己去判断的，这是别人写好的代码并且逻辑正确。但是产品提了一个新的需求：他要拉取指定时间段内的商品变动信息，这就需要手动执行定时任务。可是原来的代码没有办法做到

，怎么办？而且这个功能肯定需要一个Html页面来协助，这就少不了Controller、Service

等等。

这时候构造函数就牛逼了，我们可以通过自定义构造函数将参数传递给当前的这个Job

然后在Job类中进行判null操作，以此来区分是quartz在执行还是我去手动执行的。大致的示例代码如下：

Controller && Service：

Controller代码：

@RequestMapping(value = "funcTwo")

@ResponseBody

**public** JSONObject funcTwo(String s , String e, HttpSession session){

**return** service.funcTwo(s , e , session);

}

Service代码：

**public** JSONObject funcTwo(String s, String e, HttpSession session) {

JSONObject result = **new** JSONObject();

**new** JobGetChangeProduct (s , e).doExecute(**null**);

result.put("status", "success");

result.put("desc", "请求执行完成");

**return** result;

}

JobGetChangeProduct.java作为一个服务切入到SpringMvc的Service中。

**public** **class** JobGetChangeProduct **extends** RootJob {

**private** String startDate = **null**;

**private** String endDate = **null**;

**public** JobGetChangeProduct (String startDate, String endDate) {

**this**.startDate = startDate;

**this**.endDate = endDate;

}

**public** JobGetChangeProduct () { // quartz 调用默认构造方法

}

**public** **void** doExecute(JobExecutionContext context) {

**new** RsyncGetProductByDate(startDate , endDate);

}

}

实际上quartz调用的就是这个类的默认构造函数，所以需要显示的声明出来。

RsyncGetProductByDate.java根据时间是否为null去判断是quartz在执行还是手动执行；如果startDate和endDate为空，也就不是在手动调用定时任务，那么时间变动范围就是以前的2个小时；如果不为空，那就是你指定的时间范围，如：2016-08-01 00:00:00 到2016-09-01 00:00:00。

RsyncGetProductByDate.java判null代码如下：

**if**(StringUtils.*isNotBlank*(**this**.startDate) && StringUtils.*isNotBlank*(**this**.endDate)){

// 兼容线上运营人员的临时需求 - Yangcl

request.setChangedDateBegin(**this**.startDate);

request.setChangedDateEnd(**this**.endDate);

}**else**{

// 执行定时任务

request.setChangedDateBegin(rsyncDateCheck.getStartDate());

request.setChangedDateEnd(rsyncDateCheck.getEndDate());

}

2.2.2 构造代码块

一个比较装逼的点，实际项目中很少见到，但是面试官们总爱问。构造代码块在对象初始化的时候会**优先于构造方法执行**，目的也是为了初始化(包括字段和方法等)，但他偏向于初始化那些“共用”的内容。比如有个字段，所有的自定义构造函数都离不开他，都需要为他赋值，为了避免代码重复，就可以将这个共用的字段放到构造代码块中，从而精简代码。

具体示例如下：

**public** **class** Person {

**private** String name;

**private** Integer age;

**private** Integer eyes;

{ // 构造代码块

**this**.eyes = 2;

}

**public** Person() {

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

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

System.***out***.println("eyes = " + **this**.eyes + "\n");

}

**public** Person(String name) {

**this**.name = name;

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

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

System.***out***.println("eyes = " + **this**.eyes + "\n");

}

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

**this**.name = name;

**this**.age = age;

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

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

System.***out***.println("eyes = " + **this**.eyes);

}

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

**new** Person();

**new** Person("tom");

**new** Person("Ketarna" , 24);

}

}

测试结果如下：

name = null

age = null

eyes = 2

name = tom

age = null

eyes = 2

name = Ketarna

age = 24

eyes = 2

每个正常人都有2个眼睛，所以这里“共用”的内容就是eyes。

2.2.3 静态构造代码块

day06-06-面向对象(静态代码块).avi

2.3构造函数私有化

2.2.1 private constructor()

顾名思义就是将类的默认构造函数的public标识变成private，外部无法调用一个类的私有构造方法，不再让这个类产生对象。这样的设计多见于工具类且这个类中都是静态的方法。比较典型的就是单例模式。

2.2.2 单例模式

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。

单例模式有以下特点：

　　1、单例类只能有一个实例。

　　2、单例类必须自己创建自己的唯一实例。

　　3、单例类必须给所有其他对象提供这一实例。

* 0xde29012 双重锁定(懒汉式单例)

**public** **class** Singleton {

**private** Singleton() { // 构造函数私有化

}

**private** **static** Singleton singleton = **null**;

// 静态工厂方法 双重检查锁定

**public** **static** Singleton getInstance() {

**if** (singleton == **null**) {

**synchronized** (Singleton.**class**) {

**if** (singleton == **null**) {

singleton = **new** Singleton();

}

}

}

**return** singleton;

}

}

* 0xde29013 静态内部类(懒汉式单例)

**public** **class** Singleton {

**private** Singleton() { // 构造函数私有化

}

**private** **static** **class** LazyHolder {

**private** **static** **final** Singleton **INSTANCE** = **new** Singleton();

}

**public** **static** **final** Singleton getInstance() {

**return** LazyHolder.**INSTANCE**;

}

}

利用了classloader的机制来保证初始化instance时只有一个线程，所以线程是安全的，同时没有性能损耗，所以实际项目开发一般倾向于使用这一种。

* 0xde29014 饿汉式单例

**public** **class** Singleton {

**private** Singleton() { // 构造函数私有化

}

// 饿汉式单例类.在类初始化时，已经自行实例化

**private** **static** **final** Singleton ***single*** = **new** Singleton();

// 静态工厂方法

**public** **static** Singleton getInstance() {

**return** ***single***;

}

}

对于饿汉式单例，类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了；而懒汉式只有当调用getInstance的时候，才会去初始化这个单例。饿汉式单例天生就是线程安全的，可以直接用于多线程而不会出现问题；懒汉式本身是非线程安全的，但是这里给出的“双重锁定”和 “静态内部类”这两种线程上都是安全的。只是对于前两种来讲，双重锁定由于线程同步的问题会耗费资源，而静态内部类的解决方案在实际项目中用的更普遍。

从资源加载和性能的角度看，**饿汉式**在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成；而**懒汉式**会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，其他的与饿汉式相同。

此处参考：[JAVA设计模式之单例模式](http://blog.csdn.net/jason0539/article/details/23297037/)

2.4继承

类与类之间的继承让他们之间产生了关系，有了继承关系从而产生了多态的特性。

注意：不要为了获取其他类的功能，为了简化代码去继承；类与类之间必须存在所属关系才能去继承。

所属关系：谁是谁中的一类 or 谁是谁中的一种 or 谁是谁中的一员；他们具有一个共性特征。比如：动物的体系结构就是一个典型的继承关系，灵长类、犬科动物，猫科动物等等，他们都从属于哺乳动物。

继承特征：共性变量、方法的抽取。

Java不支持多继承

多继承隐患分析：多继承会带来安全隐患，当多个父类中定义了相同的功能(方法名)，当功能实现的业务逻辑不同时，子类对象无法确定运行哪个父类中的方法。

**class** A {

**public** **void** show(){

System.***out***.println("func a");

}

}

**class** B {

**public** **void** show(){

System.***out***.println("func b");

}

}

**class** C **extends** A , B{

}

C c = **new** C();

c.show();

2.4.1 聚集关系

2.6.2 子父类变量特点

2.6.3 子父类函数特点 – 覆盖

2.4.4 子父类构造函数特点与子类实例化过程

2.7 抽象类

2.6.1 final关键字

2.6.2模版方法模式 - final应用

2.7 接口的特点

2.8多态

2.8.1 多态与重载

2.8.2 多态与转型

* 向上转型 与 向下转型 -
* 类型提升

2.8.3 多态中成员的特点

2.9 Object类与上帝

2.9.1 equals()

2.9.2 toString()

3 内部类

3.1 内部类访问规则

3.2 内部类定义原则

3.3 静态内部类

3.4 匿名内部类

4 Java异常体系结构

4.1 异常概述

4.2 异常捕获

4.2.1异常try-catch-finally

4.2.2异常声明throws

4.2.3 throw和throws的区别

4.3 多异常处理

4.4 自定义异常

4.5 运行时异常 RuntimeException

5 String与StringBuilder

6 系统对象

day19-22-IO流(系统信息) 需要总结！

7 正则表达式

8 集合框架

9 泛型原理与应用

9.1 泛型概述

Jdk1.5以后出现的新特性，用于解决安全问题，是一个安全机制。1.5提出了三个特性：高效、简化书写和安全，泛型主要强调安全特性。泛型最普遍的应用是在集合上，为集合添加泛型，当前集合所存的元素，会在编译时期就对集合对象进行检查，将运行时其出现的问题(如异常java.lang.ClassCastException)转移到编译时期，方便问题排查。

泛型使用尖括号“<>”来描述，大括号被程序机构所使用，小括号被方法所占用，中括号被数组占用，所以就剩下尖括号了。如：

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

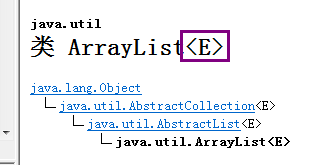
定义了一个ArrayList容器，里面存的是String类型的元素，存其他元素不行。尖括号就是用来接受类型的，和方法传递参数类似，只不过泛型传递的是类型。

集合的泛型限定是最简单的泛型应用。

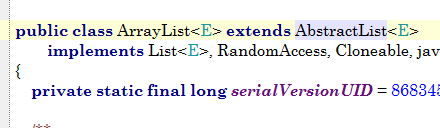
泛型对设计程序结构的时候提供了强大的扩展性，但同时也带来了局限性。

9.2泛型类

应该在什么时候定义类泛型？当一个类中的引用数据类型不确定的时候，应该通过定义类泛型来完成程序的扩展。在JDK文档中以ArrayList的定义来分析，截图如下：



接下来找到ArrayList的源代码，截图如下：



JDK的源代码最直观的说明了应该如何定义一个泛型类，以及泛型类改如何继承一个抽象的泛型类和实现一个泛型接口。

下面将以一个简单的实例来对泛型类的使用进行举例。

/\*\*

\* 定义一个泛型类

\* **@author** Yangcl

\* **@param** <K>

\*/

**public** **class** GenericClazz <K>{

**private** K k;

**public** **void** setObject(K k){

**this**.k = k;

}

**public** K getObject(){

**return** k;

}

}

/\*\*

\* 定义一个类

\* **@author** Yangcl

\*/

**public** **class** BlackMan {

**private** String type;

**private** String country;

**public** BlackMan(String type, String country) {

**this**.type = type;

**this**.country = country;

}

@Override

**public** String toString() {

**return** "BlackMan [type=" + type + ", country=" + country + "]";

}

}

/\*\*

\* 传入调用

\* **@author** Yangcl

\*/

**public** **class** GenericTest {

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

GenericClazz<BlackMan> gc = **new** GenericClazz<BlackMan>();

gc.setObject(**new** BlackMan("黑鬼", "美国黑鬼"));

BlackMan b = gc.getObject();

System.**out**.println(b.toString());

}

}

GenericClazz <K>类是最核心的，通过对他的扩展，可以延展出其他复杂功能，让程序从设计上更加健壮，泛型的出现省去了用Object类进行强转的繁琐，但由于泛型的类型限制特点也让程序的灵活性降低。

9.3 方法泛型

类泛型约束了整个对象，在灵活性上稍差，但如果是当你的类对象一初始化就必须明确整个类对象的类型，那么就必须定义类泛型，比如集合框架中的类；这也是用ArrayList来引入解析类泛型的原因。方法泛型是将泛型参数定义在方法上，将自己的泛型类型性定义与本类的泛型定义区分开，灵活性上极高。方法泛型又分为非静态方法泛型与静态方法泛型，主要是static关键字引起的。

9.3.1 非静态方法泛型

对泛型方法的使用进行举例，代码如下：

/\*\*

\* 定义一个泛型方法；**泛型放在返回值类型的前面修饰符的后面**

\* **@author** Yangcl

\* **@param** <T>

\*/

**public** **class** GenericClazz <T>{

**public** **void** show(T t){

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

}

**public** <E> **void** print(E e) {

System.**out**.println(e.toString());

}

**public** <R> **void** println(R r){

System.**out**.println(r.toString());

}

}

**public** **class** GenericTest {

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

GenericClazz<BlackMan> gc = **new** GenericClazz<BlackMan>();

gc.show(**new** BlackMan("黑鬼", "美国黑鬼"));

//gc.show("123as"); Eclipse会报错，因为与泛型类的定义冲突

gc.print("print(E e) = " + 112);

gc.print("print(E e) = " + "Hello world");

gc.print("print(E e) = " + **new** YellowMan("黄种人", "东亚地区"));

gc.println("println(R r) = " + 116);

gc.println("println(R r) = " + "Hello world");

gc.println("println(R r) = " + **new** YellowMan("黄种人", "南亚地区"));

}

}

运行结果如下：



9.3.2 静态方法泛型

静态方法不可以访问类上定义的泛型，因为**静态方法**先于类实例启动，所以也就无法访问到类上所定义的泛型。但是如果静态方法操作的应用数据类型不确定，需要用到泛型，那么就必须将泛型定义到方法上。泛型放在返回值类型的前面修饰符的后面。

**public** **class** GenericClazz <T>{

**public** **static** <R> **void** println(R r){

System.**out**.println(r.toString());

}

}

**public** **class** GenericTest {

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

GenericClazz.println("println(R r) = " + 116);

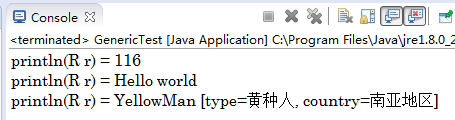
GenericClazz.println("println(R r) = " + "Hello world");

GenericClazz.println("println(R r) = " + **new** YellowMan("黄种人", "南亚"));

}

}

运行结果如下：



9.4 泛型接口

多用于web开发中Service/Dao设计。对泛型接口的使用进行举例，代码如下：

/\*\*

\* 定义一个泛型接口

\* **@author** Yangcl

\* **@param** <T>

\*/

**public** **interface** IGenericInterface<T> {

// 与接口泛型一致

**public** **void** show(T t);

// 与接口泛型不一致

**public** <E> **void** print(E e);

}

**public** **class** GenericClazz <T> **implements** IGenericInterface<T>{

@Override

**public** **void** show(T t) {

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

}

@Override

**public** <E> **void** print(E e) {

System.***out***.println(e.toString());

}

}

**public** **class** GenericTest {

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

GenericClazz<YellowMan> gc = **new** GenericClazz<YellowMan>();

gc.show(**new** YellowMan("黄种人", "中国"));

// gc.show("error generic type"); Eclipse提示错误

// The method show(YellowMan) in the type GenericClazz<YellowMan>

// is not applicable for the arguments (String)

gc.print(**new** BlackMan("黑人", "非洲"));

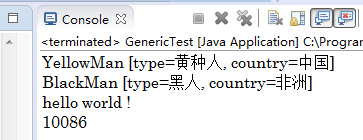
gc.print("hello world !");

gc.print(10086);

}

}

运行结果如下：



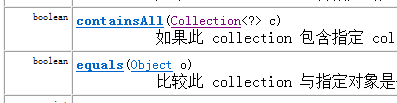
9.5 泛型高级应用 - 泛型限定

泛型限定分为向上限定和向下限定，用来约束一个方法 (或者类)所接收的泛型参数类型的范围。比如设置一个限定范围，只接收A还有A的子类型；或者只接收A还有A的父类型。前者称为向上限定，后者称为向下限定；这种泛型的限定机制源自于Java的多态性。

如果只有一个通配符，而不存在限定方向其实是没有什么意义的，这种情况等同于直接使用泛型，通常不会出现只有一个通配符的情况。

9.5.1 通配符<?>

当传入的参数类型不确定的时候，使用一个问号进行占位。在java.util.Collection<E>中的方法中使用了通配符，如下：



单独使用通配符的示例代码如下：

**public** **class** GenericTest {

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

List<String> a1 = **new** ArrayList<String>();

a1.add("小婊砸-谢婷婷");

a1.add("老娘们-谭胖子");

a1.add("小黑狗-软短虎");

List<Integer> a2 = **new** ArrayList<Integer>();

a2.add(1111);

a2.add(2222);

a2.add(3333);

print(a1);

print(a2);

}

**public** **static** **void** print(List<?> list){

Iterator<?> it = list.iterator();

**while**(it.hasNext()){

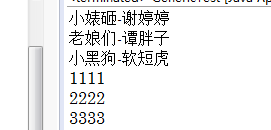
System.**out**.println(it.next());

}

}

}

运行结果如下：



泛型提高了扩展性的同时也出现了使用限制，如你无法访问length()方法：it.next().length();因为泛型是一个不确定类型，而length()方法是一个具体类型的方法。和多态中你无法预先使用子类中的具体方法是一样的，有利有弊。

向上限定、向下限定的预置代码段【9.5-1】如下：

**public** **class** Person {

**private** String name;

Person(String name){

**this**.name = "人类：" +name;

}

**public** String getName(){

**return** name;

}

}

**public** **class** YellowMan **extends** Person {

YellowMan(String name) {

**super**("黄种人：" + name);

}

}

**public** **class** BlackMan **extends** Person{

BlackMan(String name) {

**super**("黑人：" + name);

}

}

9.5.2 向上限定

<? extends A> 泛型类型限定为接收A还有A的子类型。以父类A为限制起点，所以叫向上限定。结合预置代码段【9.5-1】，使用举例如下：

**public** **class** GenericTest {

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

List<Person> a1 = **new** ArrayList<Person>();

a1.add(**new** Person("欧洲"));

a1.add(**new** Person("亚洲"));

a1.add(**new** Person("美洲"));

List<YellowMan> a2 = **new** ArrayList<YellowMan>();

a2.add(**new** YellowMan("中国"));

a2.add(**new** YellowMan("日本"));

print(a1);

print(a2);

}

// print()方法作为一个通用执行函数

**public** **static** **void** print(List<*?* ***extends*** *Person*> list){

Iterator<*?* ***extends*** *Person*> it = list.iterator();

**while**(it.hasNext()){

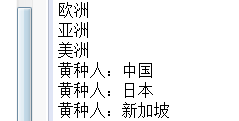
System.**out**.println(it.next().getName());

}

}

}

运行结果如下：



注意，此时it.next().getName()是可以访问到的，因为泛型的类型是明确的。

9.5.3 向下限定

<? super A> 泛型类型限定为接收A还有A的父类型。相比向上限定，向下限定的范围要小很多。结合预置代码段【9.5-1】，使用举例如下：

**public** **class** GenericTest {

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

List<Person> a1 = **new** ArrayList<Person>();

a1.add(**new** Person("欧洲"));

a1.add(**new** Person("亚洲"));

a1.add(**new** Person("美洲"));

List<YellowMan> a2 = **new** ArrayList<YellowMan>();

a2.add(**new** YellowMan("中国"));

a2.add(**new** YellowMan("日本"));

List<BlackMan> a3 = **new** ArrayList<BlackMan>();

a3.add(**new** BlackMan("肯尼亚"));

a3.add(**new** BlackMan("乍得"));

print(a1);

print(a2);

print(a3); // Eclipse提示错误如下：

// The method print(List<? super YellowMan>) in the

// type GenericTest is not applicable for the arguments

// (List<BlackMan>)

}

// print()方法作为一个通用执行函数

**public** **static** **void** print(List<? **super** YellowMan> list){

Iterator<? **super** YellowMan> it = list.iterator();

**while**(it.hasNext()){

System.**out**.println(((Person)it.next()).getName());

}

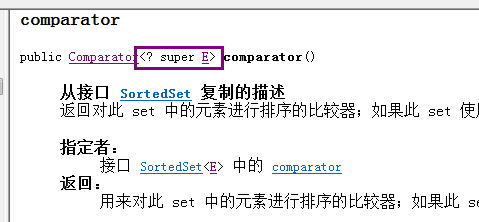
}

}

注意代码段中的红色字体部分，对比向上限定中的示例代码，这里出现了一个强制类型转换的操作！从预置代码段【9.5-1】中可以看到，YellowMan.java继承了Person.java，但是getName()方法并没有覆盖掉，所以这里要调用getName()方法就必须去找YellowMan.Java的父类。由此可以看出泛型限定的依据是“？”右边的类。

9.5.4 实际应用示例 TreeSet通用比较器

根据预置代码段【9.5-1】，细微调整，删掉其中的中文文字，设计一个TreeSet通用比较器用来说明向下限定在实际中的应用。首先在JDK文档中找到TreeSet.java。



如图所示，Comparator方法。程序设计为YellowMan创建一个TreeSet集合，为BlackMan创建一个TreeSet集合，并为他们进行排序。首先创建集合，代码如下：

**public** **class** GenericTest {

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

TreeSet<YellowMan> ySet = **new** TreeSet<>(**new** YellowComp());

ySet.add(**new** YellowMan("aken"));

ySet.add(**new** YellowMan("lisi1"));

ySet.add(**new** YellowMan("wangwu"));

ySet.add(**new** YellowMan("lisi2"));

Iterator<YellowMan> yit = ySet.iterator();

**while**(yit.hasNext()){

System.*out*.println(yit.next().getName());

}

System.*out*.println();

TreeSet<BlackMan> bSet = **new** TreeSet<>(**new** BlackComp());

bSet.add(**new** BlackMan("aobama3"));

bSet.add(**new** BlackMan("aobama2"));

bSet.add(**new** BlackMan("aobama1"));

bSet.add(**new** BlackMan("aobama4"));

Iterator<BlackMan> bit = bSet.iterator();

**while**(bit.hasNext()){

System.*out*.println(bit.next().getName());

}

}

}

第二步，创建两个比较器类，代码如下：

**import** java.util.Comparator;

**public** **class** YellowComp **implements** Comparator<YellowMan> {

**public** **int** compare(YellowMan o1, YellowMan o2) {

**return** o1.getName().compareTo(o2.getName());

}

}

**import** java.util.Comparator;

**public** **class** BlackComp **implements** Comparator<BlackMan> {

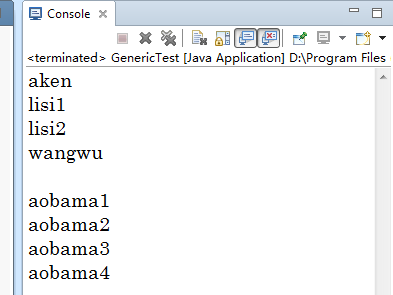
**public** **int** compare(BlackMan o1, BlackMan o2) {

**return** o1.getName().compareTo(o2.getName());

}

}

当执行GenericTest类时，结果如下：



从结果看，排序成功。但第二步创建两个比较器非常麻烦，都是多余的代码。在JDK的说明文档中明确指出了comparator()的返回值类型是这样的：<? super A>，代表Comparator接口是是一个向下限定的类型。也就是说自定义比较器的时候，只需要以父类为依据就可以，并且实现了该父类的所有子类都具有该比较器的特征。OK，新建父类比较器如下：

**import** java.util.Comparator;

**public** **class** FatherComp **implements** Comparator<Person> {

**public** **int** compare(Person o1 , Person o2) {

**return** o1.getName().compareTo(o2.getName());

}

}

修改GenericTest.java中的代码如下：

**public** **class** GenericTest {

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

TreeSet<YellowMan> ySet = **new** TreeSet<>(**new** FatherComp());

ySet.add(**new** YellowMan("aken"));

ySet.add(**new** YellowMan("lisi1"));

ySet.add(**new** YellowMan("wangwu"));

ySet.add(**new** YellowMan("lisi2"));

Iterator<YellowMan> yit = ySet.iterator();

**while**(yit.hasNext()){

System.*out*.println(yit.next().getName());

}

TreeSet<BlackMan> bSet = **new** TreeSet<>(**new** FatherComp());

bSet.add(**new** BlackMan("aobama3"));

bSet.add(**new** BlackMan("aobama2"));

bSet.add(**new** BlackMan("aobama1"));

bSet.add(**new** BlackMan("aobama4"));

Iterator<BlackMan> bit = bSet.iterator();

**while**(bit.hasNext()){

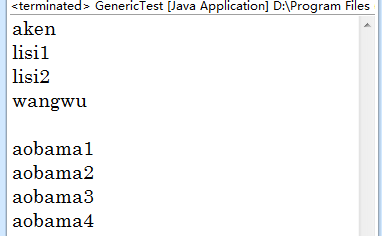
System.*out*.println(bit.next().getName());

}

}

}

执行GenericTest类结果如下：



对于上面的两个TreeSet遍历了两遍，现在对他进行抽取，形成一个通用遍历方法，这里便用到了向上限定。继续修改GenericTest.java如下：

首先修改下BlackMan.java，覆盖父类的getName()方法，来一点不一样。

**public** **class** BlackMan **extends** Person{

BlackMan(String name) {

**super**(name);

}

**public** String getName(){

**return** **super**.getName() + " black";

}

}

抽取doIterator()方法：

**public** **class** GenericTest {

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

TreeSet<YellowMan> ySet = **new** TreeSet<>(**new** FatherComp());

ySet.add(**new** YellowMan("aken"));

ySet.add(**new** YellowMan("lisi1"));

ySet.add(**new** YellowMan("wangwu"));

ySet.add(**new** YellowMan("lisi2"));

Iterator<YellowMan> yit = ySet.iterator();

doIterator(yit);

TreeSet<BlackMan> bSet = **new** TreeSet<>(**new** FatherComp());

bSet.add(**new** BlackMan("aobama3"));

bSet.add(**new** BlackMan("aobama2"));

bSet.add(**new** BlackMan("aobama1"));

bSet.add(**new** BlackMan("aobama4"));

Iterator<BlackMan> bit = bSet.iterator();

doIterator(bit);

}

**public** **static** **void** doIterator(Iterator<? **extends** Person> it){

**while**(it.hasNext()){

System.out.println(it.next().getName());

}

}

}

修改BlackMan.java是为了体现上面红色代码中，当泛型是BlackMan的时候，走的子类getName()，不再是父类了。

10 多线程同步

10.1 概述

10.1.1 进程

进程就是正在进行中的程序，每一个进程执行都有一个执行顺序，该顺序是一个执行路径，或者叫一个控制单元。线程就是进程中的一个独立的控制单元。线程在控制着进程的执行。一个进程中至少有一个线程。

10.1.2 多线程的随机性

多个线程同时去获取CPU的执行权，CPU执行到谁谁就运行。在某一个时刻(1/100000毫秒)只会有一个线程在运行(多核心CPU除外)。CPU进行快速切换，以达到程序同时运行的错觉。可以认为多线程的运行是在互相抢夺CPU的执行权，至于执行多长时间，由CPU决定。

10.2 继承Thread类

基本语法，没什么可扩展的，示例如下：

**public** **class** Demo **extends** Thread{

**public** **void** run(){

**for**(**int** x = 0 ; x < 10 ; x ++)

System.out.println("thread demo has running... " + x);

}

}

**public** **class** ThreadTest {

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

Demo demo = **new** Demo();

demo.start();

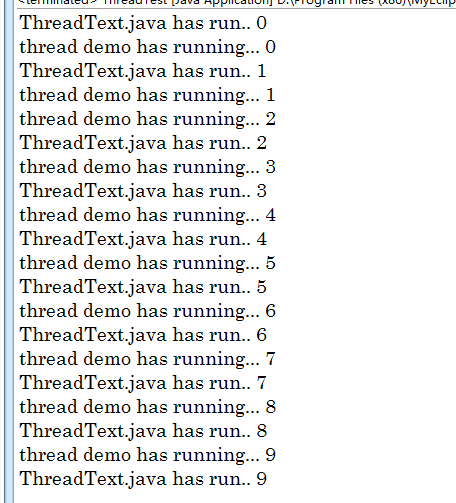
**for**(**int** x = 0 ; x < 10 ; x ++)

System.out.println("ThreadText.java has run.. " + x);

}

}

继承了Thread类则Java虚拟机认为他就是一个线程类了，与非线程类不同的是当你调用run()方法的时候需要使用该对象的start()方法。运行结果如下：



此结果显示了多线程的随机性。

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

10.3 实现Runnable接口

创建线程的另一种方法，声明实现Runnable接口，然后实现该类的run()方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递并启动。在JDK文档中：



Runnable 接口应该由那些打算通过某一线程执行其实例的类来实现。类必须定义一个称为 run 的无参数方法。Runnable 为非 Thread 子类的类提供了一种激活方式。通过实例化某个 Thread 实例并将自身作为运行目标，就可以运行实现 Runnable 的类而无需创建 Thread 的子类。大多数情况下，如果只想重写 run() 方法，而不重写其他 Thread 方法，那么应使用 Runnable 接口。这很重要，**因为除非程序员打算修改或增强类的基本行为，否则不应为该类创建子类**。示例代码如下：

**public** **class** Ticketer **implements** Runnable{

**private** **int** ticketNum = 10;

**private** **boolean** flag = **true**;

**public** **void** run() {

**while**(flag){

**if**(ticketNum > 0){

String name = Thread.currentThread().getName();

System.out.println(name + "-卖出票号：" + ticketNum--);

}**else**{

flag = **false**;

}

}

}

}

定义一个售票员类，一共10张票，在测试类中会实例化两个售票员卖这10张票。代码如下：

**public** **class** ThreadTest {

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

// 创建售票员类的对象

Ticketer ticketer = **new** Ticketer();

// 实例化两个售票员，注意他们共享ticketer对象中的10张票

Thread thread1 = **new** Thread(ticketer);

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

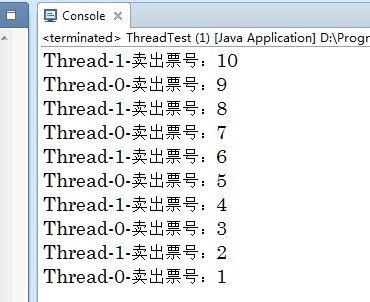
thread1.start();

thread2.start();

}

}

运行结果如下

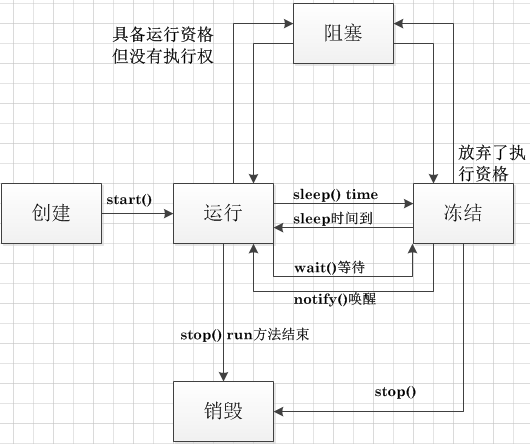


多么和谐的输出结果。那么为什么要将Runnable接口的子类对象实例作为参数传递给Thread类呢？是因为自定义的run方法所属的对象是Runnable的子类对象，而在Java中创建线程的类只有Thread.java一个；也就是说：虽然你的类实现了Runnable接口，也重写了run方法，**但是你的类不是一个线程类，因为你没有继承Thread类**，Java虚拟机不识别！所以你就必须借助Thread类来完成你自定义线程类的初始化。

这里体现出了一种多态性，Runnable接口类似于一个标识接口的作用(这种说法并不严谨，真正的标识接口是没有一个实现方法的)，让实现它的类具备一个多线程的**特征**，从而解决了单继承的局限性。

10.4 线程运行状态

线程的5种状态转化图如下：



10.5 同类型 多线程同步

同步的线程是相同的Runnable实现类。

10.5.1 线程安全分析

从10.4节中的线程运行状态的图中可以看到，当一个线程处于阻塞或者冻结状态的时候，结合10.3节中Ticketer.java的代码，可以看出当多个线程操作**同一个共享数据**的时候会出现潜在的数据操作异常。这一异常是由于线程(阻塞|冻结)和线程对CPU资源的竞争引起的。重新分析Ticketer.java源码如下：

**public** **class** Ticketer **implements** Runnable{

**private** **int** ticketNum = 10;

**private** **boolean** flag = **true**;

**public** **void** run() {

**while**(flag){

**if**(ticketNum > 0){

/\*\*

\* 假设这一时刻ticketNum = 1

\* 在下一个时刻，第一个线A程进入了，但同时也被阻塞或冻结

\* 又下一个时刻，第二个线B程进入，也被冻结

\* 又一个时刻，第三个线程C进入，也被冻结，

\* 同时前两个线程依然在冻结中，ticketNum 依然等于1

\* 然后这三个线程随机开始执行(ABC任意一个)，

\* 第一个随机线程执行完成，ticketNum=0;

\* 第二个随机线程执行完成，ticketNum=-1

\* 第三个随机线程执行完，ticketNum=-2

\*/

String name = Thread.*currentThread*().getName();

System.*out*.println(name + "-卖出票号：" + ticketNum--);

}**else**{

flag = **false**;

}

}

}

}

为了效果明显，我们将模拟四个线程同时卖票，并且再次修改Ticketer.java，对他进行冻结模拟。新代码如下：

**public** **class** Ticketer **implements** Runnable{

**private** **int** ticketNum = 10;

**private** **boolean** flag = **true**;

**public** **void** run() {

**while**(flag){

**if**(ticketNum > 0){

**try** {

Thread.sleep(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

String name = Thread.currentThread().getName();

System.out.println(name + "-卖出票号：" + ticketNum--);

}**else**{

flag = **false**;

}

}

}

}

**public** **class** ThreadTest {

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

// 创建售票员类的对象

Ticketer ticketer = **new** Ticketer();

// 实例化4个售票员，注意他们共享ticketer对象中的10张票

Thread thread1 = **new** Thread(ticketer);

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

Thread thread3 = **new** Thread(ticketer);

Thread thread4 = **new** Thread(ticketer);

thread1.start();

thread2.start();

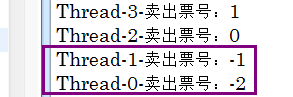
thread3.start();

thread4.start();

}

}

运行结果如下，可以看到出现了负数的情况。



10.5.2 线程同步

为了解决10.5.1中的线程安全问题，Java引入了线程同步的概念：synchronized。可用来给对象和方法或者代码块加锁。

* Ox01 方法上synchronized

**public** **synchronized** **void** dosomething(){

System.*out*.println("方法上加锁");

}

这种加锁方式简单粗暴。以去银行存钱为例子，代码如下：

/\*\*

\* 因为多个客户存钱，所以将客户定义为多线程类。

\* **@author** Yangcl

\*/

**public** **class** Customer **implements** Runnable{

**private** Bank b = **new** Bank();

**public** **void** run() {

**for**(**int** x = 1 ; x < 11 ; x ++){

b.saveMoney (100 , x);

}

}

}

/\*\*

\* 因为银行是多个客户都去存钱的地方，所以他定位在

\* 共享资源的位置。

\* **@author** Yangcl

\*/

**public** **class** Bank {

**private** **int** sum;

**public** **synchronized** **void** saveMoney(**int** n , **int** x){

// 注意！从这里取得当前执行的线程对象！

String name = Thread.currentThread().getName();

sum = sum + n ;

**try**{Thread.sleep(20);}**catch**(Exception e){}

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

}

}

模拟出四个人去存款：

**public** **class** BankDemo {

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

Customer cus = **new** Customer();

Thread t1 = **new** Thread(cus, "小婊砸");

Thread t2 = **new** Thread(cus, "谭胖子");

Thread t3 = **new** Thread(cus, "麻子祥");

Thread t4 = **new** Thread(cus, "碧池冰");

t1.start();

t2.start();

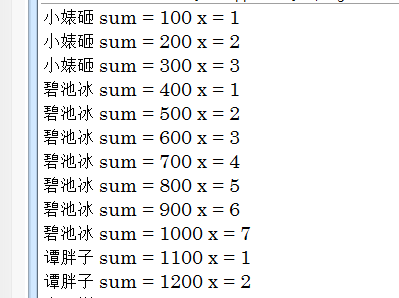
t3.start();

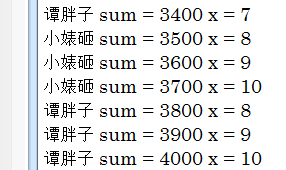
t4.start();

}

}

执行结果部分如下：





* Ox02 代码块加synchronized

**synchronized** (对象) {

System.out.println("代码块加锁");

}

这种加锁方式相对讲究，这里的对象指的是**同步锁**。尤其值得指出的是代码块加锁的方式需要你根据实际业务，仔细分析你所共享的数据是什么！确定你所共享的数据以后，将**他们**全部放入代码块中，有几个放几个。锁只有两个值：0和1。

其实，synchronized就是一个上厕所关门的情景。外面排队的人是线程，门里面是代码块，你不开门其他线程就进不来。一毫秒有多长，要看是在锁里面还是在锁外面。

没有持有锁的线程即使获取CPU的执行权也进不去，因为没有获取锁。

对代码块加synchronized举例，修改Bank.java代码如下：

**public** **class** Bank {

Object object = **new** Object();

**private** **int** sum;

**public** **void** saveMoney(**int** n , **int** x){

**synchronized**(object){

// 注意！从这里取得当前执行的线程对象！

String name = Thread.currentThread().getName();

sum = sum + n ;

**try**{Thread.sleep(20);}**catch**(Exception e){}

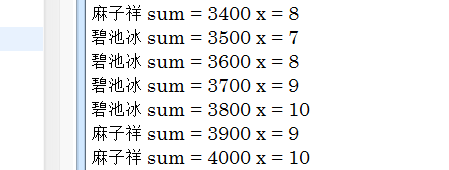
System.out.println(name + " sum = " + sum + " x = " + x);

}

}

}

运行结果部分如下：



10.5.3 同步锁类型与安全

需要指出的是synchronized修饰方法的时候，他的锁是默认的：this；当synchronized修饰代码块的时候，他的锁可以是任意一个类型。那么一个新的问题就来了，当一个类中的共享资源有两个以上的地方需要进行线程同步的时候，那么他的锁应该如何设定。

这个问题的固定解决方法是：在一个类中，同一批需要被共享的资源，当他们有两个以上的地方需要进行线程同步的时候需要保证锁的对象实例相同。如果这些地方都是用synchronized修饰方法来保证资源同步的，则无需考虑锁的对象实例是否相同；如果有的地方使用synchronized修饰方法，有的地方使用synchronized修饰代码块，那么必须让修饰代码块的地方的锁使用this，以此来保证锁的唯一性。

在上面的陈述中，如果锁不是唯一的，那么线程同步将会失败。如果锁是惟一的，则只有一个厕所，如果不唯一则有多个厕所。

(11.12)很遗憾，暂时还没有找到最恰当的例子。建议如果出现上述的情况，最好将锁都定义为this。

但如果同步方法与同步代码块修饰的是一个静态共享数据，则同步方法所使用的同步锁是该方法所在类的字节码文件对象：类名.class，所以如果同时出现同步代码块也在修饰这个静态共享数据的话，那么他的锁就是类名.class，如下：

**synchronized** (Ticketer.class) {

System.out.println("代码块加锁");

}

10.5.4 死锁

死锁是两个甚至多个线程被永久阻塞时的一种运行局面，这种局面的生成伴随着至少两个线程和两个或者多个资源。在线程引起死锁的过程中形成了一个依赖于资源的循环。**嵌套封锁**：这是死锁最主要的原因的，如果你已经有一个资源了就要避免封锁另一个资源。

* Ox01 嵌套封锁举例

代码如下：

**public** **class** DeadLock **implements** Runnable{

**private** **boolean** flag ;

**public** DeadLock(**boolean** flag) {

**this**.flag = flag;

}

**private** **static** UserInfo locka = **new** UserInfo();

**private** **static** UserMoney lockb = **new** UserMoney();

**public** **void** run() {

**if**(flag){

**synchronized** (locka) {

System.out.println("if lock a");

**synchronized** (lockb) {

System.out.println("if lock b");

}

}

}**else**{

**synchronized** (lockb) {

System.out.println("else lock b");

**synchronized** (locka) {

System.out.println("else lock a");

}

}

}

}

}

**public** **class** DeadLockTest {

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

Thread thread1 = **new** Thread(**new** DeadLock(**true**));

Thread thread2 = **new** Thread(**new** DeadLock(**false**));

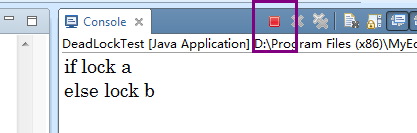
thread1.start();

thread2.start();

}

}

结果如下：

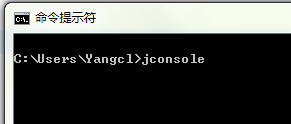


可以看到程序没有停止，陷入了互相等待的状态，进入死锁。

* Ox02 定位死锁线程

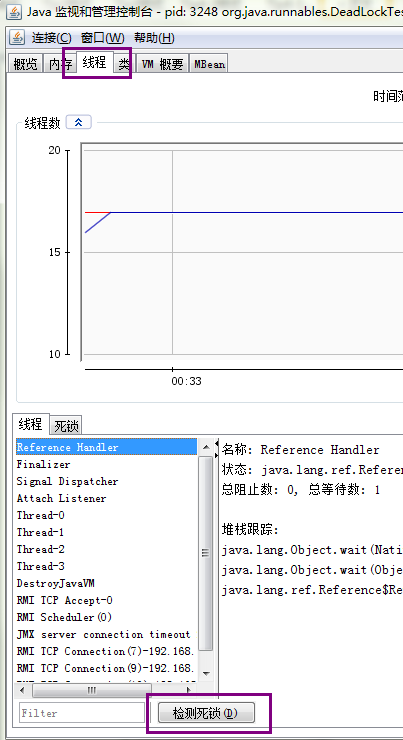
有两种方法可以确定死锁线程，1使用JDK给我们的的工具JConsole，可以通过打开cmd然后输入jconsole打开。2直接使用JVM自带的命令。

1. **JConsole 图形工具**



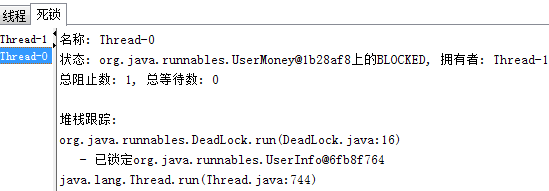


打开线程选项卡，然后点击左下角的“检测死锁”

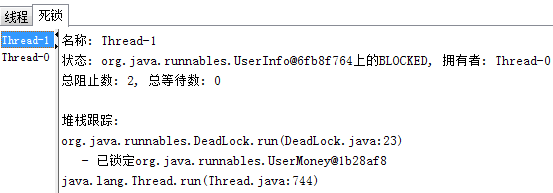


：

jconsole就会给我们检测出该线程中造成死锁的线程，点击选中即可查看详情，如【图10.5.1】和【图10.5.2】



【图10.5.1】



【图10.5.2】

从上面两幅图中我们可以看出：

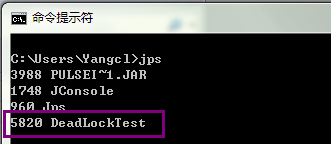
Thread-0它想申请UserMoney这个资源，但是这个资源已经被Thread-1拥有了，所以就堵塞了。Thread-1它想申请UserInfo这个资源，但是这个资源已经被Thread-0拥有了，所以就堵塞了。

Thread-0一直等待UserMoney这个资源；Thread-1一直等待UserInfo这个资源，于是这两个线程就这么僵持了下去，造成了死锁。

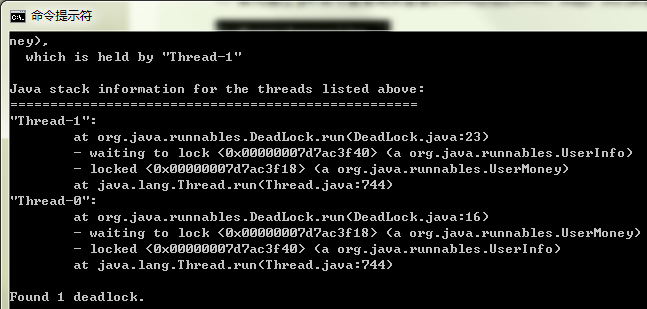
1. **JVM自带的命令**

这种方式用户体验不好，但做记录。

首先通过 jps 命令查看需要查看的Java进程的vmid，如图，我们要查看的进程DeadLockTest的vmid号是5820；



然后利用 jstack 查看该进程中的堆栈情况，在cmd中输入 jstack -l 5820，移动到输出的信息的最下面即可得到：



这段信息在输出的最后。

10.6 异类型 多线程同步(线程间通信)

可以理解为**异类多线程同步**或**线程间通信**；同步的线程是**不同的**Runnable实现类。

其实就是多个不同类型的线程在操作同一个资源，但是操作的动作不同。有的线程向资源里面存数据，有的线程从资源里面取数据。其中等待唤醒机制是最简单的一种生产者消费者模型，由两条线程来操作共享资源的存取；生产者消费者则是多条线程来操作共享资源的存取。

相比同类型多线程同步，这个内容有趣的多。异类多线程同步最重要的概念是：等待唤醒机制，它描述了一个线程去唤醒另一个线程的场景；一共只有两个线程在同步。在此基础之上，一个线程去唤醒多个线程(异类多线程允许有2个以上的Runnable实现类线程)的场景，则被称作：生产/消费者通讯模型。

在JDK1.5推出以后，极大力度的改进了多线程同步模型，出现了替代synchronized的类，原因在于synchronized对线程的控制力度不够小。

10.6.1 等待唤醒机制

等待唤醒机制主要研究的是两个线程；在他们线程安全的前提下，对同一个资源进行交替操作的过程。

Ox01 wait()/notify()深入

 wait()/notify()/notifyAll()，这三个方法都使用在线程同步中，而操作他们的则是synchronized所持有的监视器对象(即：锁)；请认真读这句话中的每个字。之所以在线程同步中使用，正式因为只有同步才具有锁。

wait()等待线程都被存在线程池中，notify()都从线程池中去找，唤醒是任意性的。在JDK中对这两个方法的解释如下：

**wait()：** public final void wait() throws InterruptedException

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。换句话说，此方法的行为就好像它仅执行 wait(0) 调用一样。当前线程必须拥有此对象监视器。该线程发布对此监视器的所有权并等待，直到其他线程通过调用 notify 方法，或 notifyAll 方法通知在此对象的监视器上等待的线程醒来。然后该线程将等到重新获得对监视器的所有权后才能继续执行。

**notify()：** public final void notify()

唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。**选择是任意性的**，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。直到当前线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。

Ox02 等待唤醒经典示例 厨师/服务员

创建共享资源类：

/\*\*

\* 这是一个共享资源：蛋糕

\* **@author** Yangcl

\*/

**public** **class** Cakes {

**private** String name;

**private** **boolean** flag = **false**;

/\*\*

\* 制作蛋糕

\* **@param** name

\*/

**public** **synchronized** **void** makeCake(String name){

**if**(flag){// 等待服务员取走蛋糕

// 这里代码未必能执行到,因为this.notify()

**try**{**this**.wait();}**catch**(Exception e){}

}

**this**.name = name;

String chefs = Thread.currentThread().getName();

System.out.println(name + "蛋糕【" + chefs + "】做完了");

flag = **true**;

**this**.notify();

}

/\*\*

\* 出售蛋糕

\*/

**public** **synchronized** **void** sellCake(){

**if**(!flag){//等待厨师做完蛋糕

// 这里代码未必能执行到,因为this.notify()

**try**{**this**.wait();}**catch**(Exception e){}

}

String waiter = Thread.currentThread().getName();

System.out.println(**this**.name + "蛋糕 ----> 【" + waiter + "】取走了" );

flag = **false**;

**this**.notify();

}

}

创建两个异类多线程类：

/\*\*

\* 这是一个做蛋糕的厨师

\* **@author** Yangcl

\*/

**public** **class** Chefs **implements** Runnable {

**private** Cakes res;

**public** Chefs(Cakes res) {

**this**.res = res;

}

**public** **void** run() {

**int** x = 0;

**for**(**int** i = 1 ; i < 11; i ++){

**if**(x==0){

res.makeCake("第" + i + "个巧克力");

}**else**{

res.makeCake("第" + i +"白奶油");

}

x = (x+1)%2;

}

}

}

/\*\*

\* 这是一个卖蛋糕的服务员

\* **@author** Yangcl

\*/

**public** **class** Waiter **implements** Runnable {

**private** Cakes res;

**public** Waiter(Cakes res) {

**this**.res = res;

}

**public** **void** run() {

**for**(**int** i = 1 ; i < 11; i ++){

res.sellCake();

}

}

}

建立测试类：

**public** **class** Launcher {

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

Cakes res = **new** Cakes();

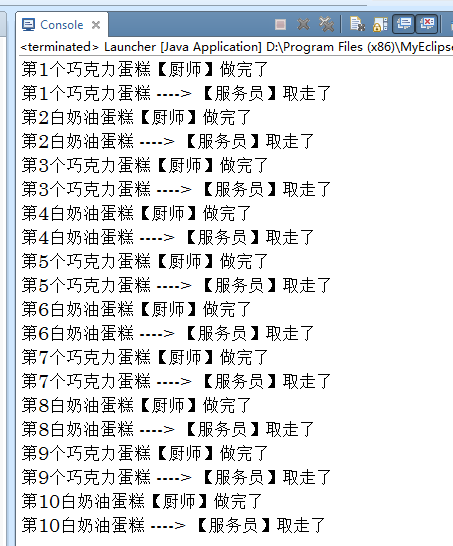
**new** Thread(**new** Chefs(res) , "厨师").start();

**new** Thread(**new** Waiter(res) , "服务员").start();

}

}

运行结果如下：



10.6.2 生产者消费者

* Ox01 错误原因分析

等待唤醒机制，它描述了一个线程去唤醒另一个线程的场景；一共只有两个线程在同步。在此基础之上，一个线程去唤醒多个线程的场景，则被称作：生产/消费者通讯模型。但是

10.6.1中的代码并不适用在生产/消费者中，例如改变Launch.java但其他类的代码不变，多加两个线程进入就会出现错误

**public** **class** Launcher {

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

Cakes res = **new** Cakes();

**new** Thread(**new** Chefs(res) , "厨师1").start();

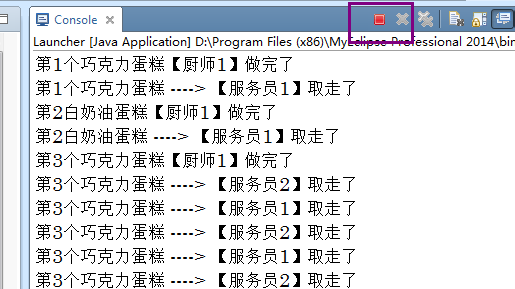
**new** Thread(**new** Chefs(res) , "厨师2").start();

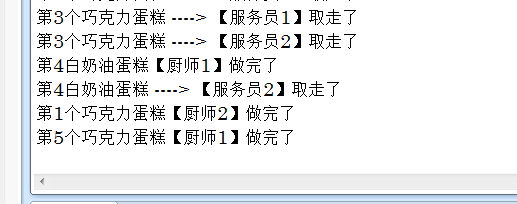
**new** Thread(**new** Waiter(res) , "服务员1").start();

**new** Thread(**new** Waiter(res) , "服务员2").start();

}

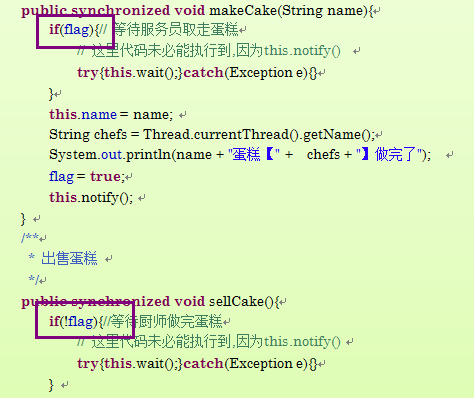
}





【图10.6.1】

从【图10.6.1】中可以看到，运行结果并没有结束，但这里的情况并不是发生了死锁，而是4个线程之间全部处于等待状态了。结合下图的代码进行分析：



第一个原因：

一个线程被wait()后，他进入了线程池等待被唤醒，线程池是一个队列结构，先进先出，而notify()方法只唤醒队列中的第一个线程。这就是等待唤醒机制中的代码在生产消费者模型中出现错误的原因之一。

第二个原因：

没有判断flag标记。notify()方法唤醒队列中的第一个线程，该线程将继续向下执行，跳出了if语句控制的代码块，而没有去判断标记是否生效；这就造成被唤醒的线程可能获得了执行资格，但他本该处于继续等待却没有等待，flag也就失效了。所以if应该改成while循环，notify()方法应该改成notifyAll();

* Ox02 生产/消费者代码模型

生产线程：放入对象；消费线程：取出对象

生产/消费者通讯模型的规则是，仅当集合中没有对象时，生产线程会放入一个对象，如有集合中有一个对象时，消费线程要马上取出这个对象

生产者线程运象一但运行，就会尝试锁定与消费者对象共享的对象集合，一但它锁定这个集合(即进入synchronized保护的代码块，如果集合中没有对象，它就会放入一个，然后发出notify()通知；如果集合中有对象，生产者线程就会调用wait()方法，将自己处于“等待”状态，直到收到一个notify()通知：

消费线程运一但运行，就会尝试锁定与生产线程对象共享的对象集合，一但它锁定这个集合(即进入synchronized保护的代码块，如果集合中没有对象，消费线程就会调用wait()方法，将自己处于“等待”状态，直到收到一个notify()通知；如果集中有对象，它将取出这个对象后，发出notify()通知告诉生产者线程，要放入数据了

等待唤醒机制中的代码改进如下：

/\*\*

\* 这是一个共享资源：蛋糕

\* **@author** Yangcl

\*/

**public** **class** Cakes {

**private** String name;

**private** **boolean** flag = **false**;

/\*\*

\* 制作蛋糕

\* **@param** name

\*/

**public** **synchronized** **void** makeCake(String name){

**while**(flag){// 等待服务员取走蛋糕

// 这里代码未必能执行到,因为this.notify()

**try**{**this**.wait();}**catch**(Exception e){}

}

**this**.name = name;

String chefs = Thread.currentThread().getName();

System.out.println(name + "蛋糕【" + chefs + "】做完了");

flag = **true**;

**this**.notifyAll();

}

/\*\*

\* 出售蛋糕

\*/

**public** **synchronized** **void** sellCake(){

**while**(!flag){//等待厨师做完蛋糕

// 这里代码未必能执行到,因为this.notify()

**try**{**this**.wait();}**catch**(Exception e){}

}

String waiter = Thread.currentThread().getName();

System.out.println(**this**.name + "蛋糕 ----> 【" + waiter + "】取走了" );

flag = **false**;

**this**.notifyAll();

}

}

/\*这是一个做蛋糕的厨师\*/

**public** **class** Chefs **implements** Runnable {

**private** Cakes res;

**public** Chefs(Cakes res) {

**this**.res = res;

}

**public** **void** run() {

**int** x = 0;

**for**(**int** i = 1 ; i < 11; i ++){

**if**(x==0){

res.makeCake("第" + i + "个巧克力");

}**else**{

res.makeCake("第" + i +"白奶油");

}

x = (x+1)%2;

}

}

}

/\*这是一个卖蛋糕的服务员\*/

**public** **class** Waiter **implements** Runnable {

**private** Cakes res;

**public** Waiter(Cakes res) {

**this**.res = res;

}

**public** **void** run() {

**for**(**int** i = 1 ; i < 11; i ++){

res.sellCake();

}

}

}

**public** **class** LauncherAll {

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

Cakes res = **new** Cakes();

**new** Thread(**new** Chefs(res) , "厨师1").start();

**new** Thread(**new** Chefs(res) , "厨师2").start();

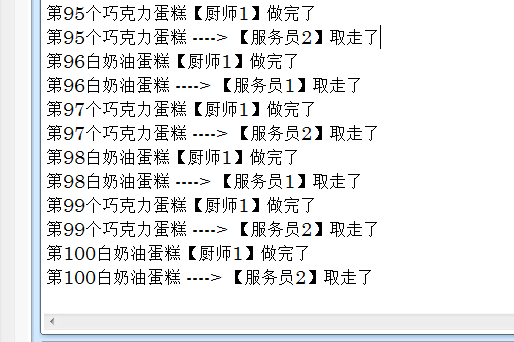
**new** Thread(**new** Waiter(res) , "服务员1").start();

**new** Thread(**new** Waiter(res) , "服务员2").start();

}

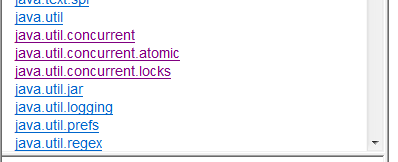
}

运行结果如下：



10.7. JDK5.0对同步的改进

里程碑级的JDK1.5。一直到这里都是在描述线程同步的所有问题，在这个版本中专门提出了一个软件包：java.util.concurrent.locks来更细粒度的解决线程同步问题。在新的技术中，synchronized / wait() / notify() / notifyAll()挂掉了，他们分别被Lock和Condition两个类替换了。



java.util.concurrent则专门针对高并发。

10.7.1 java.util.concurrent.locks.Lock

Lock 实现提供了比使用 synchronized 方法和语句可获得的更广泛的锁定操作。此实现允许更灵活的结构，可以具有差别很大的属性，可以支持多个相关的 Condition 对象。

支持多个相关的 Condition 对象，让锁的控制力度变的更加细致，变为显式控制；synchronized是一种隐式的锁操作。

newCondition()是Lock的一个常用方法，它返回绑定到此 Lock 实例的新 Condition 实例。

lock() 获取锁。

unlock() 释放锁。

10.7.2 java.util.concurrent.locks.Condition

JKD文档中描述：Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中，Lock **替代了 synchronized 方法和语句**的使用，Condition **替代了 Object 监视器方法的使用**。

由此可见：synchronized / wait() / notify() / notifyAll()挂了。

await() 造成当前线程在接到信号或被中断之前一直处于等待状态。

signal() 唤醒一个等待线程。

10.7.3 生产/消费者改进

**import** java.util.concurrent.locks.Condition;

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

**public** **class** Cakes {

**private** String name;

**private** **boolean** flag = **false**;

**private** Lock lock = **new** ReentrantLock();

**private** Condition conChefs = lock.newCondition();

**private** Condition conWaiter = lock.newCondition();

**public** **void** makeCake(String name){

lock.lock();

**try**{

**while**(flag){

conChefs.await(); // 显式让厨师线程等待

}

**this**.name = name;

String chefs = Thread.currentThread().getName();

System.**out**.println(name + "蛋糕【" + chefs + "】做完了");

flag = **true**;

conWaiter.signal(); // 显式唤醒服务员线程

}**catch**(InterruptedException e){ // TODO e}

**finally**{

lock.unlock();

}

}

**public** **void** sellCake(){

lock.lock();

**try**{

**while**(!flag){

conWaiter.await();

}

String waiter = Thread.currentThread().getName();

System.**out**.println(**this**.name + "蛋糕 ----> 【" + waiter + "】取走了" );

flag = **false**;

conChefs.signal(); // 虽然有signalAll()方法，但没必要使用了

}**catch**(InterruptedException e){ // TODO e }

**finally**{

lock.unlock();

}

}

}

**public** **class** Chefs **implements** Runnable {

**private** Cakes res;

**public** Chefs(Cakes res) {

**this**.res = res;

}

**public** **void** run() {

**int** x = 0;

**for**(**int** i = 1 ; i < 101; i ++){

**if**(x==0){

res.makeCake("第" + i + "个巧克力");

}**else**{

res.makeCake("第" + i +"白奶油");

}

x = (x+1)%2;

}

}

}

**public** **class** Waiter **implements** Runnable {

**private** Cakes res;

**public** Waiter(Cakes res) {

**this**.res = res;

}

**public** **void** run() {

**for**(**int** i = 1 ; i < 101; i ++){

res.sellCake();

}

}

}

**public** **class** LockLauncher {

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

Cakes res = **new** Cakes();

**new** Thread(**new** Chefs(res) , "厨师1").start();

**new** Thread(**new** Chefs(res) , "厨师2").start();

**new** Thread(**new** Waiter(res) , "服务员1").start();

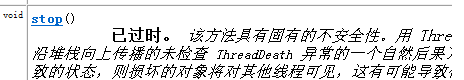
**new** Thread(**new** Waiter(res) , "服务员2").start();

}

}

10.8 停止线程

Java在1.5以前用Thread.stop()方法来停止一个处于冻结状态的线程，但在1.4以后这个方法被废弃了。



10.8.1线程停止原理

只有一种，run方法结束。开启多线程运行，运行代码通常都是循环结构，只要控制住循环，就可以让run方法结束，也就是线程结束。举例如下：

**public** **class** StopRun **implements** Runnable {

**private** **boolean** flag = **true**;

**public** **void** run() {

**int** num = 1;

String name = Thread.*currentThread*().getName();

**while**(flag){

System.***out***.println(name + " ---- 运行 " + num++ + "次");

}

System.***out***.println(name + " ---- 完犊子了");

}

**public** **void** stopRun(){

**this**.flag = **false**;

}

}

**public** **class** StopRunTest {

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

StopRun sr = **new** StopRun();

Thread t1 = **new** Thread(sr, "狗屌祥");

Thread t2 = **new** Thread(sr, "谭胖子");

t1.start();

t2.start();

**int** num = 0;

**while**(**true**){

**if**(num++ ==60){

sr.stopRun();

**break**;

}

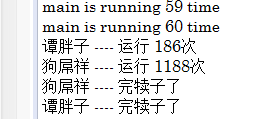
System.***out***.println("main is running " + num + " time");

}

}

}

运行结果如下：



10.8.2 特殊情况处理

如果主线程执行完了，但是t1和t2两个线程却处于**冻结**状态，也就无法在run方法的while循环体中执行，while无法执行flag = false的代码；这就意味着run方法永远无法执行完成，也就没法停止线程。

此时可以强制**中断**处于冻结状态的线程。Thread类提供了interrupt()方法执行这个操作，他的描述如下：如果线程在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法，或者该类的 join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int) 方法过程中受阻，则其**中断状态**(这里的中断状态就是**冻结**状态)将被清除，它还将收到一个InterruptedException。

需要注意的是中断状态不是停止线程。所谓**中断**即：将处于冻结状态的线程，强制恢复到运行状态中来，interrupt()是在清除冻结状态。运行状态的线程可以读取标记，也就可以执行完run方法从而停止线程。他专门解决线程被冻结但没有被唤醒(notify)的情景。

举例如下：

**public** **class** InterruptFreeze **implements** Runnable {

**private** **boolean** flag = **true**;

**public** **synchronized** **void** run() {

**int** num = 1;

String name = Thread.currentThread().getName();

**while**(flag){

**try**{

wait(); // 模拟冻结状态，注意synchronized关键字

}**catch**(InterruptedException e){

flag = **false**; // 不置为false会执行一次后继续冻结

System.**out**.println(name + "冻结了");

}

System.**out**.println(name + " ---- 运行 " + num++ + "次");

}

System.**out**.println(name + " ---- 完犊子了");

}

**public** **void** stopRun(){

**this**.flag = **false**;

}

}

**public** **class** InterruptFreezeTest {

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

InterruptFreeze interrupt = **new** InterruptFreeze();

Thread t1 = **new** Thread(interrupt, "狗屌祥");

Thread t2 = **new** Thread(interrupt, "谭胖子");

t1.start();

t2.start();

**int** num = 0;

**while**(**true**){

**if**(num++ ==60){

t1.**interrupt();**

t2.**interrupt();**

**break**;

}

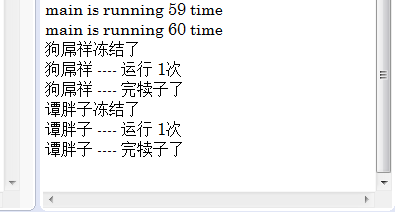
System.**out**.println("main is running " + num + " time");

}

}

}

运行结果如下：



10.9 守护线程

setDaemon()方法将一个线程标记为守护线程，所谓的守护线程就是一个后台线程。当正在运行的线程都是守护线程时，Java 虚拟机退出，该方法必须在启动线程前调用。也就是说：当所有的前台线程都结束后，后台线程就会自动结束；无论后台线程处于运行状态、阻塞状态还是冻结状态都会强制结束。

一个线程A依赖于另一个线程B，当B线程结束的时候A线程就失去了存在的意义，在这种情况下，通常会将A线程设置为守护线程。

随便举个例子如下：

**public** **class** StopRun **implements** Runnable {

**private** **boolean** flag = **true**;

**public** **void** run() {

**int** num = 1;

String name = Thread.currentThread().getName();

**while**(flag){

System.**out**.println(name + " ---- 运行 " + num++ + "次");

}

System.**out**.println(name + " ---- 完犊子了");

}

}

**public** **class** DaemonTest {

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

StopRun sr = **new** StopRun();

Thread t1 = **new** Thread(sr, "狗屌祥");

Thread t2 = **new** Thread(sr, "谭胖子");

t1.setDaemon(**true**);

t2.setDaemon(**true**);

t1.start();

t2.start();

**int** num = 0;

**while**(**true**){

**if**(num++ ==10){

**break**;

}

System.**out**.println("main is running " + num + " time");

}

}

}

运行结果如下，你会发现并没有出现【谭胖子---- 完犊子了】和【狗屌祥---- 完犊子了】这两句话，也就是说当主函数的线程结束后，线程t1和t2自动就强制结束了。也就是说：他们自动完犊子了。



10.10 Join方法

10.10.1 主线程定义

主线程和子线程是相对的。比如一个线程A在某个方法内创建了其他的线程B和C，那么A就是B和C的主线程。主线程不一定都在类的main(String args[])方法内。

10.10.2 join()实例

主线程A在向下走的时候，当他运行到了子线程B的join()方法时，意味着B线程要申请加入到运行中来，即：B线程要CPU执行权。此时CPU的执行权还在主线程手中，当B线程的join()方法执行完后，主线程也就处于冻结状态，B.join()以后的代码也就不再运行。当B线程退出后，主线程重新获得执行权，运行完后面的代码。

根据B.join()位置的不同，举两个例子来说明上面的话。

* Ox01 位置1

**public** **class** JoinTest {

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

JoinRun jr = **new** JoinRun();

Thread gdx = **new** Thread(jr, "狗屌祥");

Thread tpz = **new** Thread(jr, "谭胖子");

gdx.start();

// 此时CPU的执行权还在主线程手中

gdx.join(); // gdx线程获得CPU执行权

/\*\*

\* 执行完join方法主线程此时处于冻结状态

\* 下面的代码也就不再运行，tpz线程也没运行。

\*/

tpz.start();

**for**(**int** x = 1; x < 6 ; x++){

System.**out**.println("main is running " + x + " time");

}

}

}

**public** **class** JoinRun **implements** Runnable {

**public** **void** run() {

String name = Thread.currentThread().getName();

**for**(**int** i = 1 ; i < 6 ; i ++){

System.**out**.println(name + " ---- 运行 " + i + "次");

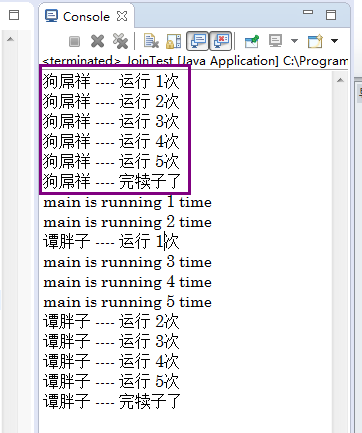
}

System.**out**.println(name + " ---- 完犊子了");

}

}

运行结果如下：

可以看到，狗吊祥线程结束后，主线程才和谭胖子线程交替抢夺CPU执行权。

* Ox02 位置2

**public** **class** JoinTest {

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

JoinRun jr = **new** JoinRun();

Thread gdx = **new** Thread(jr, "狗屌祥");

Thread tpz = **new** Thread(jr, "谭胖子");

gdx.start();

tpz.start();

gdx.join();

/\*\*

\* gdx线程获得CPU执行权，主线程被冻结。

\* 此时gdx和tpz两个线程是活的，所以他们互相竞争CPU执行权

\*/

**for**(**int** x = 1; x < 6 ; x++){

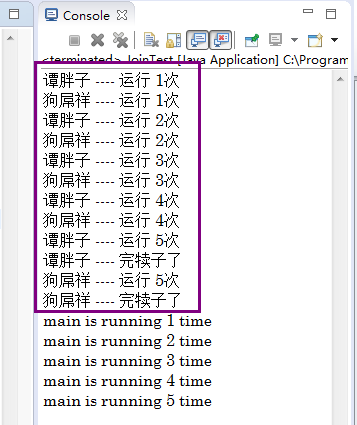
System.**out**.println("main is running " + x + " time");

}

}

}

运行结果如下：



10.11 Callable-加强型Runnable

10.11.1 Callable和Runnable有什么不同

Callable和Runnable 都代表那些要在不同的线程中执行的任务。Runnable 从 JDK1.0 开始就有了，Callable 是在 JDK1.5 增加的。它们的主要区别是 Callable 的 call() 方法可以返回值和抛出异常，而 Runnable 的 run() 方法没有这些功能。Callable 可以返回装载有计算结果的 Future 对象。

Callable是对Runnable的加强，允许你监控多线程的状态，这两个接口的源代码对比如下所示：

**public** **interface** Runnable {

**public** **void** run();

}

**public** **interface** Callable<V> {

V call() **throws** Exception;

}

从源码对比中可以得出如下结论：

1. Callable 接口下的方法是 call()，Runnable 接口的方法是 run()。
2. Callable 的任务执行后可返回值，而 Runnable 的任务是不能返回值的。
3. call() 方法可以抛出异常，run()方法不可以的。
4. 运行 Callable 任务可以拿到一个 Future 对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过 Future 对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

10.11.2 Future

**public** **interface** Future<V> {

**boolean** cancel(**boolean** mayInterruptIfRunning);

**boolean** isCancelled();

**boolean** isDone();

V get() **throws** InterruptedException, ExecutionException;

V get(**long** timeout, TimeUnit unit)

**throws** InterruptedException, ExecutionException, TimeoutException;

}

Future 定义了5个方法：

1. boolean cancel(boolean mayInterruptIfRunning)：试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 cancel() 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 mayInterruptIfRunning 参数确定是否应该以试图停止任务的方式来中断执行此任务的线程。此方法返回后，对 isDone() 的后续调用将始终返回 true。如果此方法返回 true，则对 isCancelled() 的后续调用将始终返回 true。
2. boolean isCancelled()：如果在任务正常完成前将其取消，则返回 true。
3. boolean isDone()：如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。
4. V get()throws InterruptedException,ExecutionException：如有必要，等待计算完成，然后获取其结果。
5. V get(long timeout, TimeUnit unit) throws InterruptedException ,

ExecutionException, TimeoutException;： 如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。

也就是说Future提供了三种功能：

1. 判断任务是否完成；
2. 能够中断任务；
3. 能够获取任务执行结果。

从上面两节中可以看出：

Callable和Future都无法在新线程中(new Thread(Runnable r))使用，Thread 类只支持 Runnable。不过 Callable 可以使用 ExecutorService (这个概念将在下面的线程池11.2中说明)。因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

10.11.3 FutureTask

* Ox690226 初步简介

这是新特性中非常重要的一个类，其源代码实现关系如下：

**public** **interface** RunnableFuture<V> **extends** Runnable, **Future**<V> {

**void** run();

}

**public** **class** FutureTask<V> **implements** RunnableFuture<V> {

...

}

FutureTask实现了 Runnable 和 Future，所以兼顾两者优点，既可以在 Thread 中使用，又可以在ExecutorService中使用。

* Ox690229 初级实例演示

**import** java.util.concurrent.Callable;

**public** **class** MassageCallable **implements** Callable<String> {

@Override

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

**return** "Hello world callable!";

}

}

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

MassageCallable callable = **new** MassageCallable();

FutureTask<String> task = **new** FutureTask<String>(callable);

Thread t1 = **new** Thread(task , "朱朱");

t1.start(); // 启动线程

String msg = **null**;

**try** {

msg = task.get();

} **catch** (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

System.***out***.println("msg = " + msg);

task.cancel(**false**); // 取消线程

}

FutureTask 是为了弥补 Thread 的不足而设计的，它可以让程序员准确地知道线程什么时候执行完成并获得到线程执行完成后返回的结果。FutureTask 是一种可以取消的异步的计算任务，它的计算是通过 Callable 实现的，它等价于可以携带结果的 Runnable，并且有三个状态：等待、运行和完成。完成包括所有计算以任意的方式结束，包括正常结束、取消和异常。

volatile 关键字

volatile 是一个特殊的修饰符，只有成员变量才能使用它。在Java并发程序缺少同步类的情况下，多线程对成员变量的操作对其它线程是透明的。volatile 变量可以保证下一个读取操作会在前一个写操作之后发生。线程都会直接从内存中读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

ThreadLocal 变量

ThreadLocal 是Java里一种特殊的变量。每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

11 高并发

11.1 并发同步知多少

11.1.1 串行和并行

热闹的景点买票人很多，而这时只有一个窗口售票，大家排队依次买票就可以理解为**串行**。排队人太多了，旁边又加开了几个窗口，多人在不同的窗口同时买票可以理解为**并行**。如果只能开一个窗口，这时好多着急的人围上来，有问价格的，有掏钱的，又有取票的，在这个过程中售票员在同时应对多个买票人，可以理解为**并发**。

我们经常在计算机上一边听歌一边写文档(或者处理其他的事情)，这就是一种并发行为。表面看两个程序是同时进行，为什么不是并行呢？计算机只有一个CPU所以只能支持一个线程运行。有人说:我家里计算机是多核CPU可以同时支持多个线程运行，确实是这样，但如下几点认知:

1. 虽然是多核CPU但是系统总线内存是共用的在加载内存数据时仍然需要串行访问。
2. 目前的开发语言仍然是过程型开发，暂无方法能自动的切割任务 进行并行计算。
3. 操作系统在线程调度时随着内核的增加复杂性递增，目前最多支持8核。

串行|并行|并发，如下图所示：

**时间段T**

**串行**

**并行**

**线程A**

**线程B**

**线程C**

**线程A**

**线程B**

**线程C**

**并发**

**线程A**

**线程B**

**线程C**

**线程D**

**线程E**

**线程F**

那么计算机是如何做到一边播放歌曲一边支持文档编辑呢？操作系统会把CPU的执行时间划分微妙级别的时间片段，每一个时间片内去调度一个线程执行，多个线程不断的切换执行，因此在人类可感知的时间段(秒级)内线程是同时执行的，所以多个线程在某个时间段内的同时执行就是并发。

11.1.2 共享资源的访问与串行

互联网应用基本上都是支持多用户多请求同时访问服务器端的，所以互联网应用都是支持并发的，那么高并发的主要困难是什么呢？操作系统会给每个线程分配独立的内存空间和时间片，所以线程间是隔离的。但是如果线程访问线程外的内存空间，文件系统，输入输出设备，数据库或者其他存储设备时就会发生资源竞争，**共享资源的访问必须串行**，保证串行访问资源的机制就是同步，JAVA中经常使用的同步机制有synchronized关键字,java.util.concurrent.locks.Lock系列类。

11.2 线程池

11.2.1 线程池基本概念

创建多个线程不光麻烦而且相对影响系统性能。线程池的优点如下：

1. 避免线程的创建和销毁带来的性能开销。
2. 避免大量的线程间因互相抢占系统资源导致的阻塞现象。
3. 能够对线程进行简单的管理并提供定时执行、间隔执行等功能。

Java里面线程池的顶级接口是 Executor，不过真正的线程池接口是 ExecutorService， ExecutorService 的默认实现是 ThreadPoolExecutor；普通类 Executors 里面调用的就是 ThreadPoolExecutor。

**public** **interface** Executor {

**void** execute(Runnable command);

}

**public** **interface** ExecutorService **extends** Executor {

**void** shutdown();

List<Runnable> shutdownNow();

**boolean** isShutdown();

**boolean** isTerminated();

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

...

}

**public** **class** Executors {

**public** **static** ExecutorService newCachedThreadPool() {

**return** **new** ThreadPoolExecutor(

0,

Integer.MAX\_VALUE, 60L,

TimeUnit.SECONDS,

**new** SynchronousQueue<Runnable>()

);

}

...

}

* Ox90AE26 Executors 提供四种线程池：

1. newCachedThreadPool 是一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute() 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。注意，可以使用 ThreadPoolExecutor 构造方法创建具有类似属性但细节不同（例如超时参数）的线程池。
2. newSingleThreadExecutor 创建是一个单线程池，也就是该线程池只有一个线程在工作，所有的任务是串行执行的，如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
3. newFixedThreadPool 创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小，线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
4. **newScheduledThreadPool** 创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。

* Ox90AE27 ThreadPoolExecutor构造函数参数

通过 ThreadPoolExecutor 的构造函数，撸一撸线程池相关参数的概念：

**public** ThreadPoolExecutor(

**int** corePoolSize,

**int** maximumPoolSize,

**long** keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory) {

**this**(corePoolSize, maximumPoolSize, keepAliveTime, unit,

workQueue, threadFactory, defaultHandler);

}

参数含义如下：

1. corePoolSize：线程池的核心线程数，一般情况下不管有没有任务都会一直在线程池中一直存活，只有在 ThreadPoolExecutor中的方法 allowCoreThreadTimeOut(

boolean value) 设置为 true 时，闲置的核心线程会存在超时机制，如果在指定时间没有新任务来时，核心线程也会被终止，而这个时间间隔由第3个属性 keepAliveTime 指定。

1. maximumPoolSize：线程池所能容纳的最大线程数，当活动的线程数达到这个值后，后续的新任务将会被阻塞。
2. keepAliveTime：控制线程闲置时的超时时长，超过则终止该线程。一般情况下用于非核心线程，只有在ThreadPoolExecutor 中的方法allowCoreThreadTimeOut

(boolean value) 设置为 true时，也作用于核心线程。

1. unit：用于指定 keepAliveTime 参数的时间单位，TimeUnit 是个 enum 枚举类型，常用的有：

TimeUnit.HOURS(小时)；

TimeUnit.MINUTES(分钟)；

TimeUnit.SECONDS(秒)；

TimeUnit.MILLISECONDS(毫秒)；

1. workQueue：线程池的任务队列，通过线程池的 execute(Runnable command) 方法会将任务 Runnable 存储在队列中。
2. threadFactory：线程工厂，它是一个接口，用来为线程池创建新线程的。

* Ox90AE28 线程池的关闭

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdown() 和 shutdownNow()。

|  |  |
| --- | --- |
| 方法名 | 功能描述 |
| shutdown() | 不会立即的终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务。 |
| shutdownNow() | 立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务。 |

* Ox40BC60 什么是Executor框架?

通过上述3组概念定义可以看出：Executor 框架是一个根据一组执行策略调用、调度、执行和控制的异步任务的框架。无限制的创建线程会引起应用程序内存溢出，所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用 Executor 框架可以非常方便的创建一个线程池。

11.2.2 线程池示例

* Ox923103a 入门样例

针对线程池的应用，这里以一个入门级别的程序进行说明，包括如何用和其运行规则等。

**class** AqiRunable **implements** Callable<String>{

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

String tname = Thread.*currentThread*().getName();

System.***out***.println("AqiRunable 线程call()方法被调用 - " + tname);

Thread.*sleep*(19000);

**return** "AqiRunable 线程返回结果 - 处理 19 秒";

}

}

**class** WeatherRunnable **implements** Callable<String>{

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

String tname = Thread.*currentThread*().getName();

System.***out***.println("WeatherRunnable 线程call()方法被调用 - " + tname);

Thread.*sleep*(11000);

**return** "WeatherRunnable 线程返回结果 - 处理 11 秒";

}

}

**class** FunkRunnable **implements** Callable<String>{

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

String tname = Thread.*currentThread*().getName();

System.***out***.println("FunkRunnable 线程call()方法被调用 - " + tname);

Thread.*sleep*(14000);

**return** "FunkRunnable 线程返回结果 - 处理 14 秒";

}

}

**class** BiterRunnable **implements** Callable<String>{

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

String tname = Thread.*currentThread*().getName();

System.***out***.println("BiterRunnable 线程call()方法被调用 - " + tname);

Thread.*sleep*(12000);

**return** "BiterRunnable 线程返回结果 - 处理 12 秒";

}

}

首先创建四个多线程类，让他们分别实现Callabel接口，并返回一个String字符串。让他们分别休眠11秒到19秒不等，用来模拟程序处理内部任务的时间。当Future对象取值的时候，这个时间还可以显示出整个代码的执行顺序。一个采用了多线程的方法，最后的执行总时间，取决于耗时最长的那个子线程的执行时间。下面给出测试代码：

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

**long** start = System.*currentTimeMillis*();

ExecutorService executor = Executors.*newCachedThreadPool*();

Future<String> aqi = executor.submit(**new** AqiRunable());

Future<String> wea = executor.submit(**new** WeatherRunnable());

Future<String> funk = executor.submit(**new** FunkRunnable());

Future<String> biter = executor.submit(**new** BiterRunnable());

executor.shutdown();

**for**(**int** i = 0 ; i < 5 ; i ++){

System.***out***.println("i = " + i);

}

System.***out***.println("\n\n等待程序处理完成... \n" );

**try** {

System.***out***.println(wea.get()); // 耗时 11 秒

System.***out***.println(biter.get()); // 耗时 12 秒

System.***out***.println(aqi.get()); // 耗时 19 秒

System.***out***.println(funk.get()); // 耗时 14 秒

} **catch** (InterruptedException | ExecutionException e) {

}

**long** end = System.*currentTimeMillis*();

System.***out***.println("耗时：" + (end - start) + " 毫秒\n");

**for**(**int** k = 0 ; k < 5 ; k ++){

System.***out***.println("k = " + k);

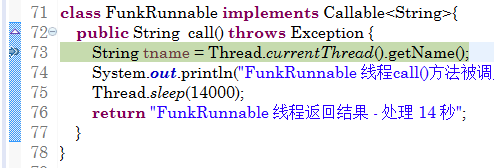
}

}

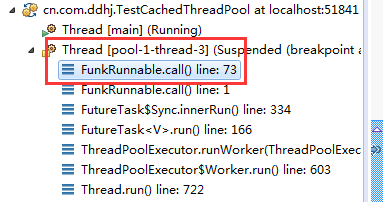
executor对象几乎在一瞬间执行完成，并且在主线程内启动了4个子线程，主线程不会等待子线程执行完而是继续向下执行，将i打印5遍。直到在try catch体内取值的时候才会因为子线程没有执行完成而卡住。为了演示多线程之间的执行不互相干扰，将在Funk

Runable线程处打一个断点来作说明。

断点截图如下：



控制台断点截图如下：



红框上面：Thread[main](Running) 显示的是执行体为主线程，Running代表其状态为运行中。红框则显示了线程名称：pool-1-thread-3，以及调用的方法。此时Console的打印信息如下：

AqiRunable 线程call()方法被调用 - pool-1-thread-1

WeatherRunnable 线程call()方法被调用 - pool-1-thread-2

BiterRunnable 线程call()方法被调用 - pool-1-thread-4

i = 0

i = 1

i = 2

i = 3

i = 4

等待程序处理完成...

WeatherRunnable 线程返回结果 - 处理 11 秒

BiterRunnable 线程返回结果 - 处理 12 秒

AqiRunable 线程返回结果 - 处理 19 秒

现在放开断点，控制台显示信息如下：

AqiRunable 线程call()方法被调用 - pool-1-thread-1

WeatherRunnable 线程call()方法被调用 - pool-1-thread-2

BiterRunnable 线程call()方法被调用 - pool-1-thread-4

i = 0

i = 1

i = 2

i = 3

i = 4

等待程序处理完成...

WeatherRunnable 线程返回结果 - 处理 11 秒

BiterRunnable 线程返回结果 - 处理 12 秒

AqiRunable 线程返回结果 - 处理 19 秒

FunkRunnable 线程call()方法被调用 - pool-1-thread-3

FunkRunnable 线程返回结果 - 处理 14 秒

耗时：728708 毫秒

k = 0

k = 1

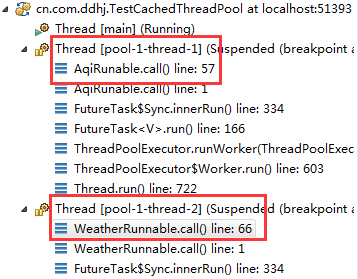
k = 2

k = 3

k = 4

可见子线程的执行没有受到影响，只是在Future对象取值的时候会出现等待，因为这里是主线程在调用子线程的返回结果，所以才会出现这种情况。

当在多个子线程打断点的时候，断点控制台如下图：



去除断点后的完整执行结果如下：

AqiRunable 线程call()方法被调用 - pool-1-thread-1

WeatherRunnable 线程call()方法被调用 - pool-1-thread-2

FunkRunnable 线程call()方法被调用 - pool-1-thread-3

BiterRunnable 线程call()方法被调用 - pool-1-thread-4

i = 0

i = 1

i = 2

i = 3

i = 4

等待程序处理完成...

WeatherRunnable 线程返回结果 - 处理 11 秒

BiterRunnable 线程返回结果 - 处理 12 秒

AqiRunable 线程返回结果 - 处理 19 秒

FunkRunnable 线程返回结果 - 处理 14 秒

耗时：19002 毫秒

k = 0

k = 1

k = 2

k = 3

k = 4

* 实际项目应用

11.3 并发编程模型

11.3 无阻塞算法

11.4 阿姆达尔定律

Java1.7新增的多线程类与方法

线程池http://www.oschina.net/question/565065\_86540

12 同步阻塞式I/O与应用

You Must Know：

1. 不论字符流还是字节流，读写都是相对内存来讲。读入到内存；从内存写入到磁盘等等。
2. **明确**源和目的。

源：输入流，InputStream | Reader

目的：输出流，OutputStream | Writer

1. **明确**操作数据是否是纯文本。

是：字符流

否：字节流

1. **明确**设备来区分使用哪个具体的流对象。

源设备：内存、硬盘、键盘

目的设备：内存、硬盘、控制台

12.1 字符编码演进

在早期的IO当中，只有字节流，没有字符流，全部都是10101010…这样的二进制数据，但是需要表示生活中的字母、数字、标点符号的时候这些二进制数据是没有可读性的。所以美国人定义了第一代的编码表：ASCII(American Standard Code for Information Interchange，美国标准信息交换代码)。这张表的作用是：把生活中的字符、数字和符号分别指定一个特定的二进制数字，比如：小写的a对应十进制是97，对应的二进制是1100001。

ASCII一共有128个。

美国人做完这个表，计算机可以显示其日常生活中的文字和符号了，但是中国的汉字远大于128个，所以中国人也做了一张码表：GB2312。收录几千个常用汉字，等于扩展了ASCII表。但是中国民族众多，有藏语，朝鲜语，维吾尔族语等等，还有一些不长用的汉字，所以又进行了一次对GB2312的扩展，形成了GBK码表(中文Windows系统默认编码表)。

与此同时，日本、泰国、希腊、德国等等等国家也都形成了自己的编码表。这种情况出现后国际标准化组织开了个会，把各国的编码表统一收录在一张新的码表中，形成了Unicode编码表，每个字符都占2字节，即汉字输入法中的全角。

但是像ASCII表中的这些内容完全用1字节就可以表示，没必要用2字节，所以国际标准化组织再次对Unicode编码表进行优化，形成了Utf-8这个通用码表。

12.1.1 字节(Byte)

字节是通过网络传输信息（或在硬盘/内存中存储信息）的单位。

1个字节等于8位二进制：1 Byte = 8 bit （比特）或这样理解：1字节（Byte）= 8位（bit）1千字节（KB,Kilobyte）=1024字节（2的10次方字节）（ 1KB = 1024 B ）

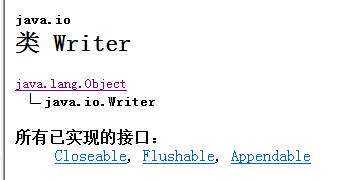
12.1.2 字符

人们使用的记号，抽象意义上的一个符号，如：'1', '中', 'a', '$', '￥' 等等。

字节是计算机中存储数据的单元，一个8位的二进制数，是一个很具体的存储空间如：0x01, 0x45, 0xFA；英文一个字母是一字节。中国字比较复杂，1字节=8位，8位从0000 0000到1111 1111，即2的8次方只能表示256个字符，2个字节是2\*8=16位，从0000 0000 0000 0000 到 1111 1111 1111 1111可以表示65535个字符。

12.2 字符流

12.2.1 写入字符流 Writer



写入字符流的抽象类。BufferedWriter, CharArrayWriter, FilterWriter,

OutputStreamWriter, PipedWriter, PrintWriter, StringWriter。在这些常用子类中，BufferedWriter、OutputStreamWriter和PrintWriter三个子类最常用。

1. **OutputStreamWriter**

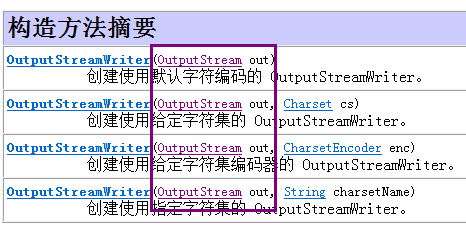
OutputStreamWriter是**字符流通向字节流的桥梁**：可使用指定的charset将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

每次调用write()方法都会导致在给定字符（或字符集）上调用编码转换器。在写入底层输出流之前，得到的这些字节将在缓冲区中累积。可以指定此缓冲区的大小，不过，默认的缓冲区对多数用途来说已足够大。注意，传递给write()方法的字符没有缓冲。

**为了获得最高效率，可考虑将OutputStreamWriter包装到BufferedWriter中，以避免频繁调用转换器**。例如：

Writer out = **new** BufferedWriter(**new** OutputStreamWriter(System.out));

OutputStreamWriter提供的构造函数需要你提供一个OutputStream对象，这个参数是必须的，其中第四个构造函数是开发中最常用的。



OutputStream是字节流对象的基类，所以OutputStreamWriter也成为**转换流**对象。

1. **BufferedWriter**

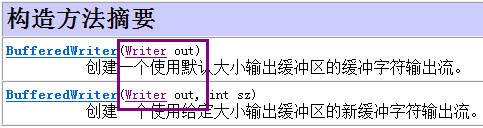
将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

可以指定缓冲区的大小，或者接受默认的大小。在大多数情况下，默认值就足够大了。该类提供了newLine()方法，它使用平台自己的行分隔符概念，此概念由系统属性line.separator定义。并非所有平台都使用新行符('\n')来终止各行。因此调用此方法来终止每个输出行要优于直接写入新行符。

通常Writer将其输出立即发送到底层字符或字节流。除非要求提示输出，否则**建议用BufferedWriter包装所有write()操作可能开销很高的Writer**(如FileWriters和OutputStreamWriters)。例如，

PrintWriter out = **new** PrintWriter(**new** BufferedWriter(**new** FileWriter("foo.out")));

将缓冲PrintWriter对文件的输出。如果没有缓冲，则每次调用print()方法会导致将字符转换为字节，然后立即写入到文件，而这是极其低效的。其构造函数如下：



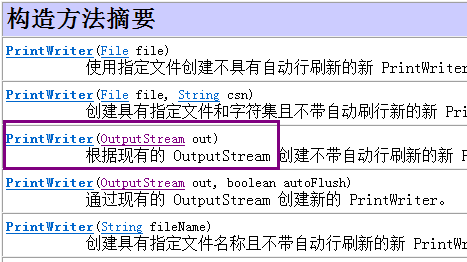
1. **PrintWriter**

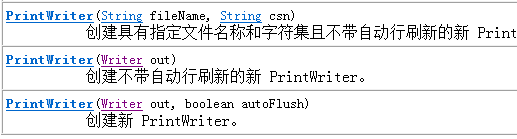
向文本输出流打印对象的格式化表示形式。此类实现在PrintStream中的所有print方法。它不包含用于写入原始字节的方法，对于这些字节，程序应该使用未编码的字节流进行写入。

与PrintStream类不同，如果启用了自动刷新，则只有在调用println、printf或format的其中一个方法时才可能完成此操作，而不是每当正好输出换行符时才完成。这些方法使用平台自有的行分隔符概念，而不是换行符。

此类中的方法不会抛出I/O异常，尽管其某些构造方法可能抛出异常。客户端可能会查询调用checkError()是否出现错误。

在以Tomcat为服务器的JavaWeb开发中，**Struts、SpringMVC里面大量使用了PrintWriter去向浏览器或者服务器传递字符流**。





* Ox01 OutputStreamWriter创建文件

将OutputStreamWriter包装到BufferedWriter中，以避免频繁调用转换器，代码如下：

**import** java.io.BufferedWriter;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.OutputStreamWriter;

**import** java.io.Writer;

**public** **class** OutputStreamWriterDemo {

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

Writer out = **null**;

FileOutputStream fos = **null**;

OutputStreamWriter osw = **null**;

**try** {

fos = **new** FileOutputStream("D:\\OutputStreamWriterDemo.txt");

osw = **new** OutputStreamWriter(fos , "utf-8");

out = **new** BufferedWriter(osw);

out.write("Hello world ! OutputStreamWriter ");

out.flush();

}

**catch** (IOException e) {

System.out.println("文件创建或写入异常");

}**finally**{

**try** {

**if**(fos != **null**)

fos.close();

**if**(osw != **null**)

osw.close();

**if**(out != **null**)

out.close();

} **catch** (IOException e) {

System.out.println("文件关闭异常");

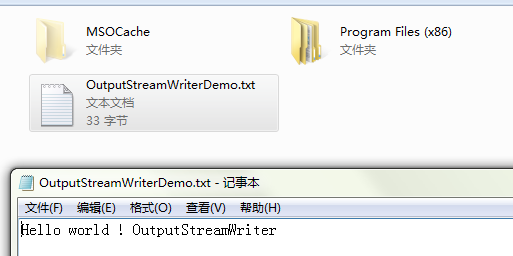
}

}

}

}

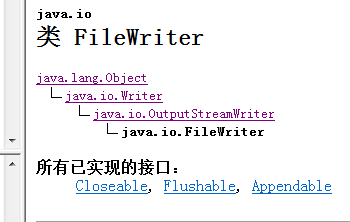
结果如下：



OutputStreamWriter创建文件只是一种应用举例，如果单纯为了创建一个文件，有更好的办法，OutputStreamWriter有一个直接子类：FileWriter。



对于FileWriter，如下：



用来写入字符文件的便捷类。FileWriter用于写入字符流。要写入原始字节流，请考虑使用FileOutputStream。

* Ox02 FileWriter创建文件

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** WriteDemo {

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

FileWriter fw = **null**;

**try** {

fw = **new** FileWriter("D:\\FileWriter.txt");

fw.write("Hello world ! File Writer ");

}

**catch** (IOException e) {

System.*out*.println("文件创建或写入异常");

}**finally**{

**try** {

**if**(fw != **null**)

fw.close();

} **catch** (IOException e) {

System.*out*.println("文件关闭异常");

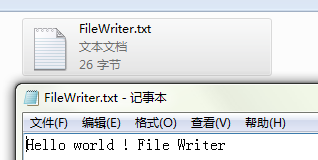
}

}

}

}

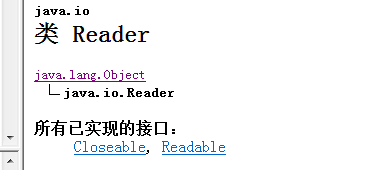
结果如下：



这种方式创建一个文件最为简洁。FileWriter(String fileName, boolean append)，根据给定的文件名以及指示是否附加写入数据的boolean值来构造FileWriter对象。如果为true则不会覆盖原有的文件。也就是说：如果文件存在，则会在这个文件内容的后面继续写入。

缺点：只能以系统默认的编码格式来创建文本文件。

12.2.2 读取字符流 Reader

用于读取字符流的抽象类。

直接已知子类：

BufferedReader , CharArrayReader , FilterReader, InputStreamReader, PipedReader, StringReader。

常用类：

**BufferedReader , InputStreamReader , FileReader**



1. **BufferedReader**
2. **InputStreamReader**
3. **FileReader**

* Ox208501 FileReader 读取文本文件

这种读取方式简单快捷，是专门用来读取文本文件的类，但也有他的缺点：无法设定读取文件的字符集。比如：我的项目是UTF-8的，但是Txt文档是以GBK保存的内容，这时候就会出现读取乱码的问题。示例代码如下：

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** FileReaderDemo {

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

FileReader fr = **null**;

/\*\*

\* 定义一个字符数组，用于存储读到的字符

\* read(char[] cbuf)返回的是读到字符的个数

\*/

**char**[] buf = **new** **char**[1024];

**try** {

fr = **new** FileReader("D:\\FileReader.txt");

**int** num = 0;

**while**((num = fr.read(buf)) != -1){

// 注意：这里不应该用println()方法。

System.out.print(**new** String(buf , 0 , num));

}

} **catch** (FileNotFoundException e) {

System.out.println("file not found...");

}**catch** (IOException e) {

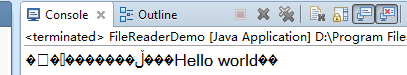
System.out.println("fr.read(buf) got exception...");

}

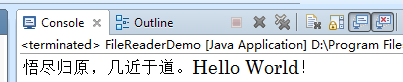
}

}

文档是GBK字符集，结果如下：



文档是UTF-8字符集，结果如下：



* Ox208502 InputStreamReader 读取文本文件

**import** java.io.BufferedReader;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.UnsupportedEncodingException;

**public** **class** InputStreamReaderDemo {

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

FileInputStream fis = **null**;

InputStreamReader isr = **null**;

BufferedReader in = **null**;

**try** {

fis = **new** FileInputStream("D:\\FileReader2.txt");

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

in = **new** BufferedReader(isr);

String line = **null**;

**while**((line = in.readLine()) != **null**){

System.out.println(line);

}

} **catch** (FileNotFoundException e) {

System.out.println("FileInputStream exception");

} **catch** (UnsupportedEncodingException e) {

System.out.println("InputStreamReader exception");

} **catch** (IOException e) {

System.out.println("readLine() exception");

}**finally**{

**try** {

**if**(fis != **null**)

fis.close();

**if**(isr != **null**)

isr.close();

**if**(in != **null**)

in.close();

}**catch** (Exception e){

System.out.println("close exception");

}

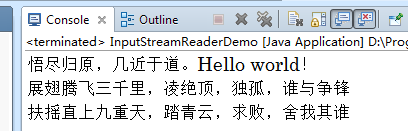
}

}

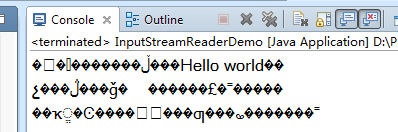
}

测试文档：FileReader.txt|GBK编码；FileReader2.txt|UTF-8编码

当采用UTF-8编码读取FileReader2.txt文件时候，结果如下：



当采用UTF-8编码读取FileReader.txt文件时候，结果如下：



代码方式虽然繁琐一点，但是非常灵活。

12.2.3 字符读写综合：文本文件复制及原理

* Ox01 基本文件拷贝

**public** **class** FileCopy {

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

**new** FileCopy().baseFileCopy();

}

**public** **void** baseFileCopy(){

FileReader fr = **null**;

FileWriter fw = **null**;

**try**{

fr = **new** FileReader("D:\\FileCopySource.txt");

fw = **new** FileWriter("E:\\FileCopySource-Copy.txt");

// 连接读写流的关键数组

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

**int** len = 0;

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

System.*out*.println("读取的字符数：" + len);

fw.write(cbuf, 0, len);

}

}**catch**(IOException e){

**throw** **new** RuntimeException("程序读写失败！");

}**finally**{

**if**(fr != **null**){

**try**{

fr.close();

}**catch**(IOException e){

}

}

**if**(fw != **null**){

**try** {

fw.close();

} **catch** (IOException e) {

}

}

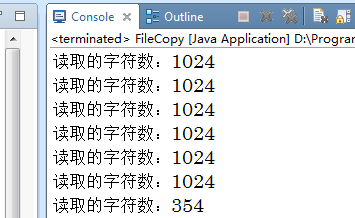
}

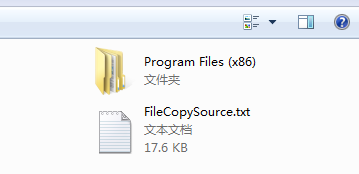
}

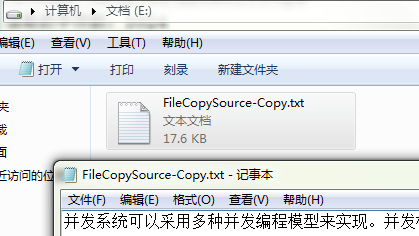
}

这种采用FileReader和FileWriter来进行文件拷贝的操作是最基本的一种方式。

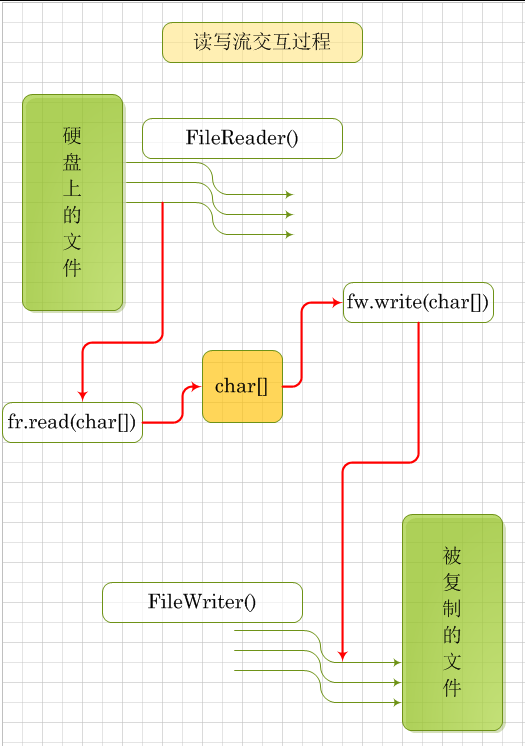
**char**[] cbuf = **new** **char**[1024]; 他是连接读写流的一个关键数组，在原理图中会提及他。程序运行结果如下：







* Ox02 读写流交互原理



Char数组就是一个缓冲池，FileReader将流注入到这个缓冲池中，FileWriter在将缓冲池中的流拿走。这幅图涵盖了所有的阻塞式IO流的读写交互原理。

* Ox03 转换流文件拷贝

这仅仅是个Demo。将BufferedReader和BufferedWriter进行一次结合使用。

**public** **class** FileCopy {

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

**new** FileCopy().bufferedFileCopy();

}

**public** **void** bufferedFileCopy(){

BufferedReader bReader = **null**;

BufferedWriter bWriter = **null**;

**try** {

bReader = **new** BufferedReader(**new** InputStreamReader(

**new** FileInputStream("D:\\FileCopySource.txt"),"utf-8"));

bWriter = **new** BufferedWriter(**new** OutputStreamWriter(

**new** FileOutputStream("F:\\FileCopySource-Copy.txt") , "utf-8"));

String line = **null**;

**int** count = 1;

**while**((line = bReader.readLine()) != **null**){

bWriter.write(line);

bWriter.newLine();

bWriter.flush();

System.*out*.println("第：" + count++ + " 行");

}

} **catch** (IOException e) {

**throw** **new** RuntimeException("程序读写失败！");

}**finally**{

**try** {

**if**(bReader != **null**)

bReader.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException("BufferedReader关闭失败！");

}**finally**{

**if**(bWriter != **null**){

**try** {

bWriter.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException("BufferedWriter关闭失败！");

}

}

}

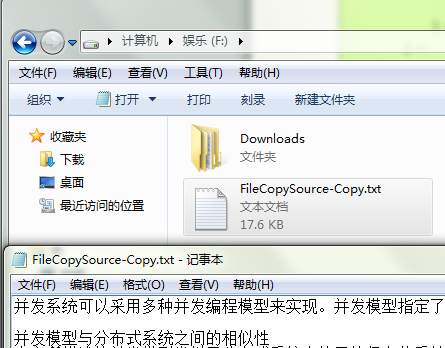
}

}

}

运行结果如下：



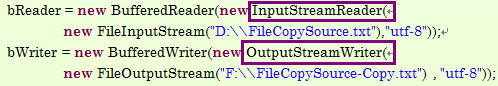


12.3 字节流

字符流的套装类：Reader.java Writer.java

1. **FileInputStream.java** -> 符节共用
2. InputStreamReader.java
3. FileReader.java extends InputStreamReader.java
4. BufferedReader.java
5. **FileOutputStream.java** -> 符节共用
6. OutputStreamWriter.java
7. FileWriter.java extends OutputStreamWriter.java
8. BufferedWriter.java

红色与蓝色的往往套装使用，如下图：

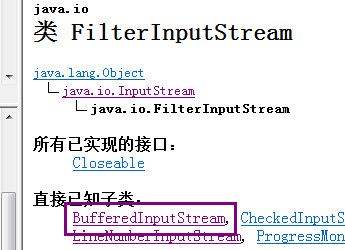


字节流的套装类：InputStream.java OutputStream.java

1. **FileInputStream.java** -> 符节共用
2. BufferedInputStream.java
3. **FileOutputStream.java** -> 符节共用
4. BufferedOutputStream.java

字符流是对字节流的装饰增强，通过强制查询编码表从而对字节流进行了一次扩展。FileInputStream.java和FileOutputStream.java是字符流与字节流都会频繁使用的类。

值得指出的是BufferedInputStream.java和BufferedOutputStream.java的继承扩展关系如下：



与字符流不同的是，字节流省去了转换流的过度。

12.3.1 字节流读取基础方式

一般都与缓冲流进行结合使用，很少用这种方式，因为效率很低。示例代码如下：

**public** **class** BasicStreamDemo {

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

FileInputStream fis = **new** FileInputStream("D:\\InputStreamDemo.txt");

FileOutputStream fos = **new** FileOutputStream("D:\\binary.txt");

**int** ch = 0;

**while**((ch=fis.read()) != -1){

System.out.println((**char**)ch + "=" +ch + "=" +Integer.toBinaryString(ch));

fos.write(ch);

fos.flush();

}

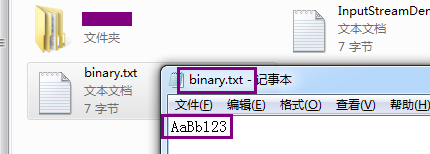
fis.close();

fos.close();

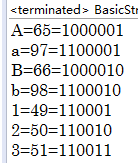
}

}

运行结果如下：



控制台信息如下：

 FileInputStream中的read()方法从此输入流中读取一个数据字节。也就是说，他在文件中读取的是一个二进制数据。

12.3.2 字节流读取之缓冲流

仅仅是个参考示例。

**import** java.io.BufferedInputStream;

**import** java.io.BufferedOutputStream;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**public** **class** BufferedStreamDemo {

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

BufferedInputStream bis = **null**;

BufferedOutputStream bos = **null**;

**try** {

bis = **new** BufferedInputStream(**new** FileInputStream("D:\\Demo.jpg"));

bos = **new** BufferedOutputStream(**new** FileOutputStream("D:\\binary.png"));

**int** bufSize = bis.available();// 创建一个大小刚好的缓冲区,大文件慎用!

System.**out**.println("文件字节大小：" + bufSize);

**byte**[] byteArr=**new** **byte**[bufSize];

**while**(bis.read(byteArr) != -1){

bos.write(byteArr);

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

System.**out**.println("BufferedInputStream read() exception ");

}**finally**{

**try** {

**if**(bis != **null**)

bis.close();

} **catch** (IOException e) {

}**finally**{

**try** {

**if**(bos != **null**)

bos.close();

} **catch** (IOException e) {

}

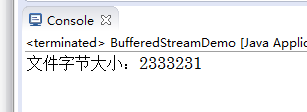
}

}

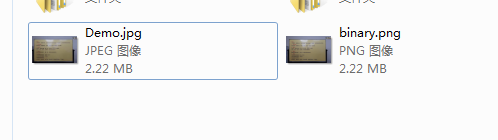
}

}

运行结果如下：



文件复制结果：



**int** bufSize = bis.available(); 是大文件分割复制时必须使用的，如果复制大音频文件，直接创建一个那么大的缓冲数组内存就溢出了。Java虚拟机默认大小是64M，建议参考值小于这个值。

12.3.3 读取键盘录入

比较典型的一类玩具，这里仅作收集。

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**public** **class** SystemInDemo {

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

BufferedReader br = **null**;

**try** {

br = **new** BufferedReader(**new** InputStreamReader(System.**in**));

String line = **null**;

**while**( (line=br.readLine()) != **null**){

**if**("over".equals(line))

**break**;

System.**out**.println(line.toUpperCase());

}

} **catch** (IOException e) {

System.**out**.println("buffered reader readline exception ");

}**finally**{

**try** {

**if**(br != **null**)

br.close();

} **catch** (IOException e) {

System.**out**.println("buffered reader close exception ");

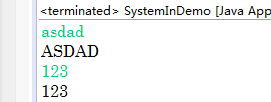
}

}

}

}

运行结果如下：



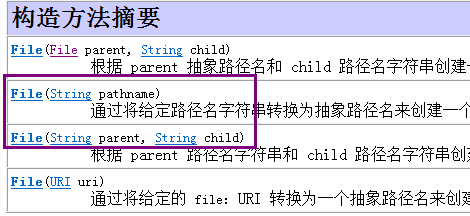
12.4 File对象精解

12.4.1 File.java概述

IO流中的重要概念，用来描述文件的一系列属性信息。Java将文件和文件夹封装成了对象的形式，即：File.java。File对象可以作为参数传递给流的构造函数，这是一种流与File的综合应用；在日常开发中很少有单独使用流来操作文件的情况，他们都是结合使用，因为IO流对象无法操作目录，也无法判断文件是否存在，更无法判断文件的在操作系统中的各种状态。

File的出现弥补了流对象的不足。流对象能够操作文件，但是流对象无法操作文件夹，也无法操作文件的属性信息(比如：一个文件是可读的还是可写的，流是不知道的)，流对象的缺点是只能操作数据，无法操作文件，而想要操作被数据封装成的那个文件的信息就只能使用File对象。

* Ox01 构造函数



* Ox02 跨平台分隔符

static String separator与系统有关的默认名称分隔符，为了方便，它被表示为一个字符串。对于UNIX平台，绝对路径名的前缀始终是 "/"。相对路径名没有前缀。表示根目录的绝对路径名的前缀为 "/" 且名称序列为空。对于Microsoft Windows平台，包含盘符的路径名前缀由驱动器号和一个 ":" 组成。如果路径名是绝对路径名，还可能后跟 "\\"。

通常这样使用：

String s\_ = File.separator;

File file = new File("C: " + s\_ + "user" + s\_ + "local" + s\_ +"hello.txt");

目的是节省代码空间。

12.4.2 文件基本操作

* Ox01 文件创建

|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
| createNewFile | boolean | 在指定位置创建文件，如果该文件已经存在，则不创建并返回false。和输出流不同，输出流对象已建立就会穿件文件；而且如果文件已经存在则会覆盖该文件。 |
| createTempFile | static File | createTempFile(String prefix, String suffix)：在默认临时文件目录中创建一个空文件，使用给定前缀和后缀生成其名称。  createTempFile(String prefix, String suffix, File directory)  是其重载形式，创建一个临时文件并指定创建到哪个目录下。 |
| mkdir() | boolean | 创建指定的目录。使用频率非常高的一个方法。 |
| mkdirs() | boolean | 创建多层目录。mkdir()方法一次只能创建一个目录，不能创建多个目录，如果尝试创建多个目录应该使用这个方法。 |
| 注! | 针对mkdir()和mkdirs()这两个方法，既然是创建就必然会涉及到去判断。在File类中有两个方法经常和他们联合使用：isDirectory()和isFile()。 | |
| renameTo() | boolean | 重新命名此抽象路径名表示的文件。renameTo(File dest)方法类似于剪切，比如将C:\\A.txt文件renameTo到D盘下，会删除C盘的文件，在底盘创建文件。假如是一个比较大的视频文件，这个过程会耗费较长时间。 |
|  |  |  |
|  |  |  |
|  |  |  |

* Ox02 文件删除

|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
| delete() | boolean | 删除失败返回false |
| deleteOnExit() | void | 在程序退出时候删除指定文件。这种删除是强制性的，如果需求必须删除一个文件，但在代码执行过程中会出现异常情况，即使将delete方法放入到finally块中也会出现无法删除的情况。因为当一个文件被系统占用的时候，你是无法删除它的。 |
|  |  |  |
|  |  |  |

* Ox03 文件判断

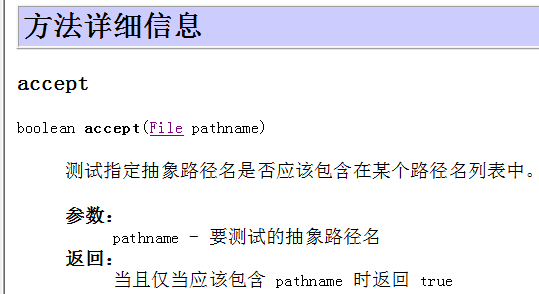
|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
| canExecute() | boolean | 判断一个文件(或应用程序)是否可以执行。当判断出一个文件是可执行文件，如果在同时调用Java.Runtime对象(Runtime.exec())，那么就可以将你电脑中的所有可执行程序全部打开。 |
| canRead() | boolean | 判断应用程序是否可以读取 |
| canWrite() | boolean | 判断应用程序是否可以修改 |
| compareTo() | int | 按字母顺序比较两个抽象路径名。 |
| exists() | boolean | 判断文件或目录是否存在，使用频率最高的方法。这个方法通常和mkdir()方法一起使用。 |
| isDirectory() | boolean | 判断是否是目录。 |
| isFile() | boolean | 判断是否是文件。 |
| 注! | 1. isDirectory()方法和isFile()方法在实际开发中去做判断的时候有一个坑：当一个文件不存在的时候，如果你用这两个方法去判断，那么都会返回false。所以，他们使用的时候是有一个共同的前提的：这个文件必须存在！否则这两个判断是没有意义的。 2. 不要通过一个名称就简单的认为他是一个文件或者文件夹。比如：   File file = new File("fileDemo.java"); 实际开发中，这个fileDemo.java有可能是一个文件夹，而不是文件。因为没有硬性的规定，文件夹一定不能以后缀名结尾。同理，文件也不一定都有后缀名，这样的文件在操作系统盘符下多了去了。 | |
| isHidden() | boolean | 判断一个文件是否是隐藏文件。注意！java对隐藏文件没有访问权限，如果你强行读取会出现异常。但是，实 际开发中这个方法的确使用的并不是很频繁。 |
| compareTo() | int | 按字母顺序比较两个抽象路径名。 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

* Ox04 获取文件信息

|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
| getName() | String | 返回文件或目录的名称。 |
| getPath() | String | 返回一个路径名字符串。 |
| getParent() | String | 返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回 null。 |
| lastModified() | long | 返回文件最后一次被修改的时间。 |
| length() | long | 返回文件的长度(大小)。BufferedInputStream类的available()方法与length()方法相似，但是available()返回的是一个int值，如果文件过大也就存不下了，返回值会出错。Length()方法返回的是一个long值，范围足够大。 |
| listRoots() | File[] | 列出可用的文件系统根。对于windows系统，会列出如C:\\  D:\\ E:\\等等。也就是操作系统的盘符 |
| list() | String[] | 返回一个字符串数组，包括目录中的文件和目录。 |
| listFiles() | File[] | 返回一个File类型的数组，这个数组包括文件和文件夹。 |
| listFiles() | File[] | 以FileFilter作为参数，这是一个文件过滤器，返回指定类型的数组，包括文件或文件夹。此方法重点演示如何重写文件过滤器。 |
| listFiles() | File[] | FilenameFilter，以文件名过滤器作为参数，返回指定类型的数组，包括文件或文件夹。此方法重点演示。 |
| 注意！listRoots()如果和list()、listFiles()结合使用，则能够遍历出盘符中的所有文件；包括系统中的隐藏文件。 | | |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

12.4.3 文件过滤器FileFilter





一个简单的接口，只有一个实现方法。

File.listFiles(FileFilter filter)方法依据FileFilter对象的accept()方法的返回值来判断这个文件是不是需要的文件。当且仅当该名称应该包含在文件列表中时返回true否则返回false。

**public** **class** FileFilterDemo {

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

FileFilterDemo fileutil = **new** FileFilterDemo();

File[] files = fileutil.suffixFilter("D:\\tgc\\controller\\coupon", ".jpg");

**if**(files != **null** && files.length != 0){

**for** (File f : files) {

System.***out***.println(f.getName());

}

}

}

**public** File[] suffixFilter(String dir, String suffix) {

File file = **new** File(dir);

**if** (!file.isDirectory()) {

System.***out***.println("No directory provided");

**return** **null**;

}

File[] files = file.listFiles(**new** FileFilter() {

**public** **boolean** accept(File pathname) {

**return** pathname.getName().endsWith(suffix);

}

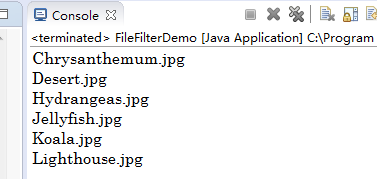
});

**return** files;

}

}

结果如下：



12.4.4文件名过滤器FilenameFilter



又是一个接口，实现此接口的类实例可用于过滤器文件名。



File.list(FilenameFilter filter)方法和File.listFiles(FilenameFilter filter)方法都是依据FilenameFilter对象的accept()方法的返回值来判断这个文件是不是需要的文件。当且仅当该名称应该包含在文件列表中时返回true否则返回false。

对于accept()方法的这两个参数也是比较有意思的，下面通过一段代码来说明。

* Ox266802 accept参数特点

**private** **static** **void** filenameFilterTest(){

File dirs = **new** File("D:\\tgc\\controller\\coupon");

String[] array = dirs.list(**new** FilenameFilter(){

@Override

**public** **boolean** accept(File dir, String name) {

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

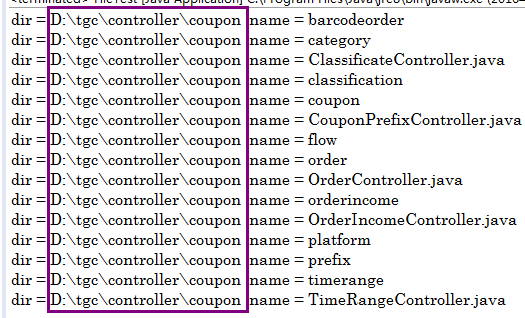
**return** **false**;

}

});

}

我以匿名内部类的方式来实现FilenameFilter接口，这段代码在运行以后将会打印出他所过滤的所有内容，重点就是dir这个参数。运行结果如下：



可以看到，紫色部分就是File构造函数中的参数，如果不传值，他是有默认值的。

* Ox396802 accept()实现

通过实现accept()方法，将会尝试去过滤出以.java结束的所有文件，示例代码如下：

**private** **static** **void** filenameFilterTest(){

File dirs = **new** File("D:\\tgc\\controller\\coupon");

String[] array = dirs.list(**new** FilenameFilter(){

@Override

**public** **boolean** accept(File dir, String name) {

**return** name.endsWith(".java");

}

});

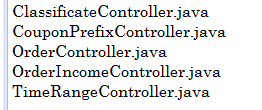
**for**(String str : array){

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

}

}

运行结果如下：



12.4.5 遍历磁盘文件

**import** java.io.File;

**public** **class** Recursion {

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

File file = **new** File("D:\\tgc");

*showDir*(file, 0);

}

**public** **static** **void** showDir(File dir , **int** level){

System.***out***.println(*getLevel*(level) + dir.getName());

level ++;

File[] files = dir.listFiles();

**for**(**int** x = 0 ; x < files.length ; x++){

**if**(files[x].isDirectory()){

*showDir*(files[x], level);

}**else**{

// 这里如果直接使用files[x]将打印路径+文件名。

System.***out***.println(*getLevel*(level) + files[x].getName());

}

}

}

**private** **static** String getLevel(**int** level){

StringBuilder sb = **new** StringBuilder();

sb.append("|--");

**for**(**int** x = 0 ; x < level ; x ++){

sb.insert(0, "| ");

}

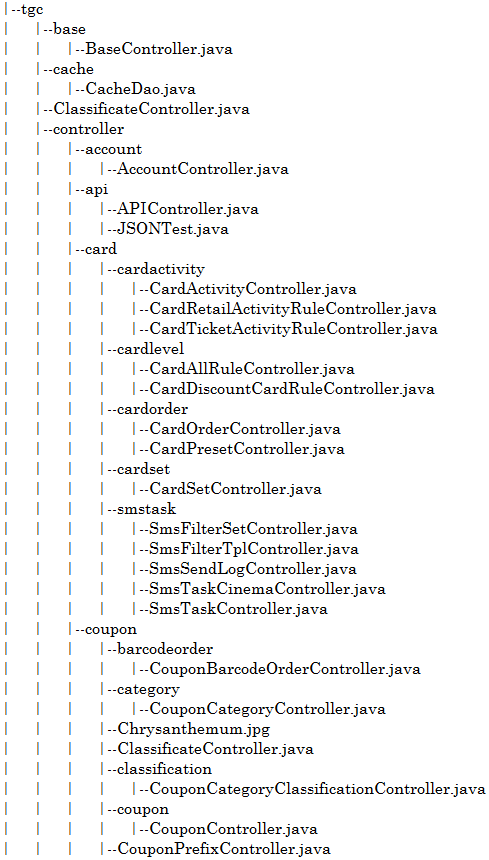
**return** sb.toString();

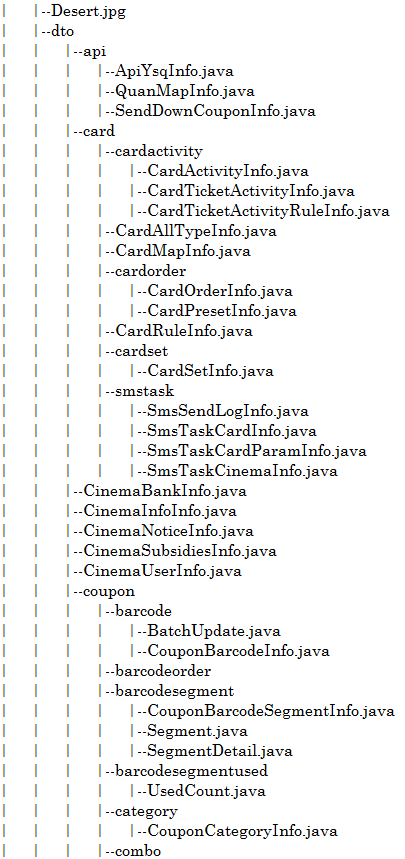
}

}

当你入侵一台服务器的时候，这个方法简直是天赐利刃。遍历出服务器的所有文件、目录，持久化到txt文件中或者发送到你指定的邮箱，Linux的系统文件配置结构将一览无余。同时，Linux系统中没有Windows系统的盘符概念。

运行结果如下：





下一节(12.4.6)将对控制台上的内容进行持久化。

12.4.6 流File综合 – 目录递归持久化

下面的代码将尝试列出C盘下的所有文件信息，并持久化到txt文件中去。代码如下：

/\*\*

\* **@descriptions** 将目标路径下的文件夹和文件持久化到一个txt文件中。

\*

\* **@param** sources 目标路径

\* **@param** target txt文件生成位置

\* **@author** Yangcl

\* **@date** 2016年4月24日-下午5:29:55

\* **@version** 1.0.0.1

\*/

**public** **static** **void** contentInit(File sources , File target){

**if**(!sources.isDirectory()){

**throw** **new** RuntimeException("目标路径必须是目录");

}

**if**(target.exists()){

target.delete();

}

**try**{

target.createNewFile();

}**catch** (IOException e) {

String name = target.getName();

**throw** **new** RuntimeException("目标文件 " + name + " 创建失败失败");

}

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

List<String> list = *showDir*(sources, 0, contentList);

BufferedWriter bWriter = **null**;

**try** {

bWriter = **new** BufferedWriter(**new** OutputStreamWriter(**new** FileOutputStream(target) , "utf-8"));

**for**(String str : list){

bWriter.write(str);

bWriter.newLine();

}

bWriter.flush();

} **catch** (IOException e) {

target.deleteOnExit();

**throw** **new** RuntimeException("程序读写失败");

}**finally**{

**if**(bWriter != **null**){

**try** {

bWriter.close();

} **catch** (IOException e) {

target.deleteOnExit();

**throw** **new** RuntimeException("BufferedWriter关闭失败");

}

}

}

}

/\*\*

\* **@descriptions** 返回一个树形层次的集合

\* **@param** dir 目标路径

\* **@param** level 树形层次记录

\* **@param** list 返回的集合

\* **@return** list

\* **@author** Yangcl

\* **@date** 2016年4月24日-下午7:03:33

\* **@version** 1.0.0.1

\*/

**private** **static** List<String> showDir(File dir , **int** level , List<String> list){

**if**(level == 0 && dir.getName().trim().length() == 0){

list.add(*getLevel*(level) + dir); // 列出盘符下所有文件

}**else**{

list.add(*getLevel*(level) + dir.getName());

}

level ++;

File[] files = dir.listFiles();

**if**(files != **null**){

**for**(**int** x = 0 ; x < files.length ; x++){

**if**(files[x].isDirectory()){

*showDir*(files[x], level , list);

}**else**{

// 这里如果直接使用files[x]将打印路径+文件名。

list.add(*getLevel*(level) + files[x].getName());

}

}

}

**return** list;

}

**private** **static** String getLevel(**int** level){

StringBuilder sb = **new** StringBuilder();

sb.append("|--");

**for**(**int** x = 0 ; x < level ; x ++){

sb.insert(0, "| ");

}

**return** sb.toString();

}

// 测试如下

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

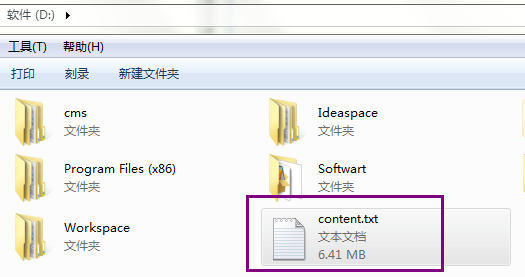
File sources = **new** File("C:\\");

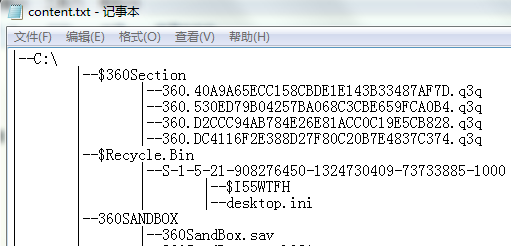
File target = **new** File("D:\\content.txt");

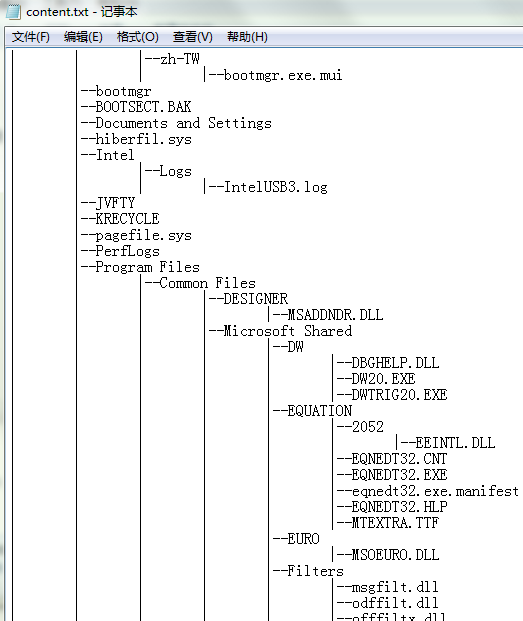
*contentInit*(sources, target);

}

运行结果如下：







C盘下的目录信息量较大，所生成的txt文档有6M。

12.5 PrintWriter 打印流

在原来的基础上需要结合Socket中的部分内容来综合分析。

12.6 文件的合并与切割

多个源对应一个目的文件或者流

12.7 对象的序列化

12.8 工具流对象

13 异步非阻塞Non-block I/O

<http://ifeve.com/java-nio-all/>

14 Socket网络通信

14.1 基本网络概念

14.1.1 IP地址

标识一台物理主机在网络节点中的位置。如果把物理主机比喻成一座大楼，那么IP就是这座大楼的地址；如果把程序比喻为大楼中的每个房间，那么端口号就是每个房间的号码。

比较特殊的IP地址：

127.0.0.1 代表本机IP localhost；

192.168.1.(0到255) 最常用的局域网保留地址段；

去查：

1 地址段是什么意思？如果局域网不在同一个地址段能不能通信？比如A在10.10.1.1，B在192.168.1.2，他们能通信吗？

2 子网掩码的作用，默认网关的作用

3 DNS服务器的作用，如何拦截恶意网站，host文件的作用

Ox01 InetAddress类对IP地址的简单操作

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**public** **class** IpDemo {

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

InetAddress inet = InetAddress.getByName("www.baidu.com");

System.out.println("主机地址：" + inet.getHostAddress());

System.out.println("主机名称：" + inet.getHostName());

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

// 获取一组主机信息

InetAddress[] inets = InetAddress.getAllByName("www.baidu.com");

**for**(InetAddress inet\_ : inets){

System.out.println("主机地址：" + inet\_.getHostAddress());

System.out.println("主机名称：" + inet\_.getHostName());

}

}

}

结果如下：



14.1.2 端口

一台电脑中可以安装很多程序，有一些网络程序可以相互之间进行通信，比如飞秋、QQ；数据要发送到对方电脑指定的网络程序上去，就必须标识这些网络应用程序。现代计算机中采用**数字**对每一个网络应用程序进行标识，这个数字就叫做端口(也称为逻辑端口或端口号)。

比如：<http://172.0.0.1:8080/> 代表你要访问的是本机的Tomcat程序。

端口号范围：0到65535

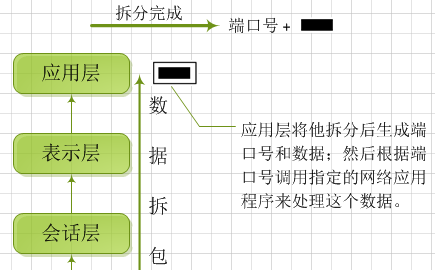
14.1.3 网络层次模型

有2种主流的网络层次模型，一种是老的：OSI(Open System Interconnection开放网路模型)参考模型；另一种是新的TCP/IP参考模型。

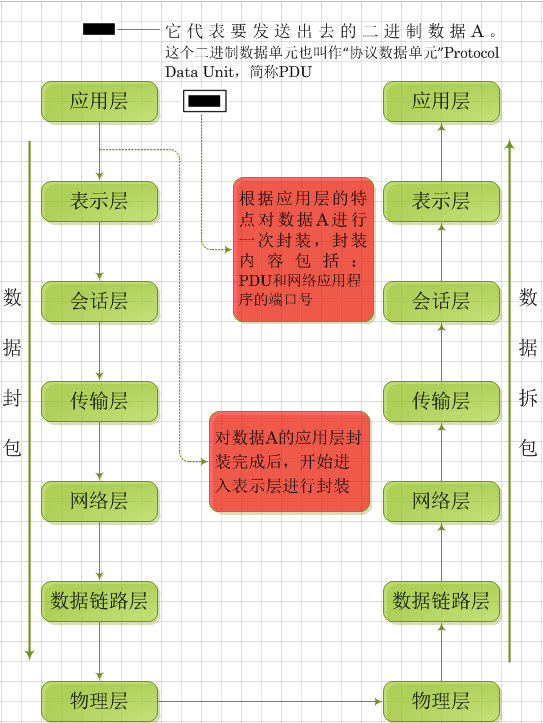


主要以OSI参考模型进行详细分析。TCP协议和UDP协议在传输层；网络层则对数据包进行IP封装。所以在Java的Socket网络通信这一技术模块中，将会主要集中在传输层和网络层进行开发。JavaWeb开发中的技术和问题则集中在应用层，应用层最主要的协议是Ftp协议、Http协议和Https协议。

值得注意的是，一个数据包的接受者当他在应用层进行数据的最后拆包的时候，他会看这个数据到底走哪个端口，换言之：这个数据分发给哪个网络应用程序来读取！如下图所示：

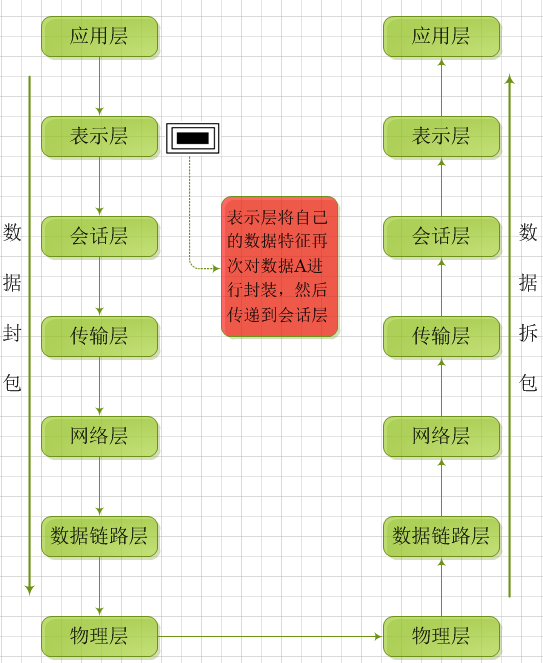


应用层

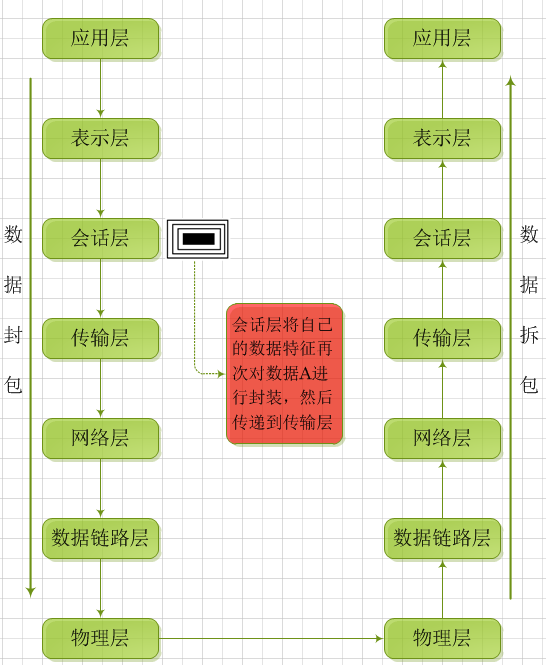


对二进制数据A追加应用层的数据特征，然后传递到表示层，继续封装表示层的数据特征。应用层的主要协议是Http协议，JavaWeb开发也是围绕这一层展开。

表示层

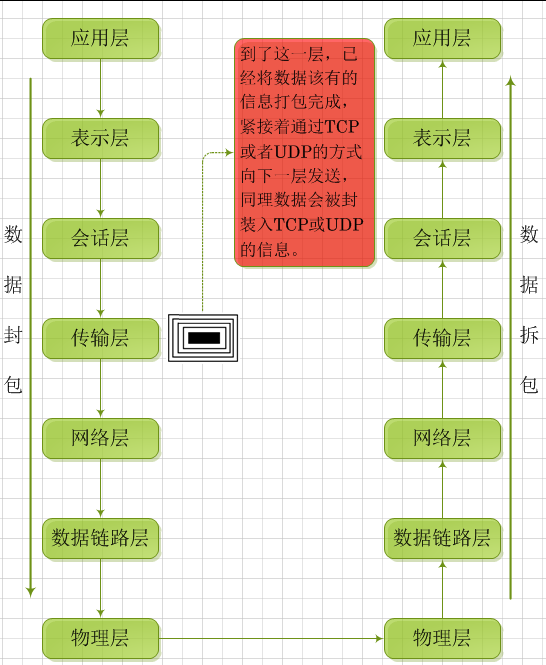


会话层

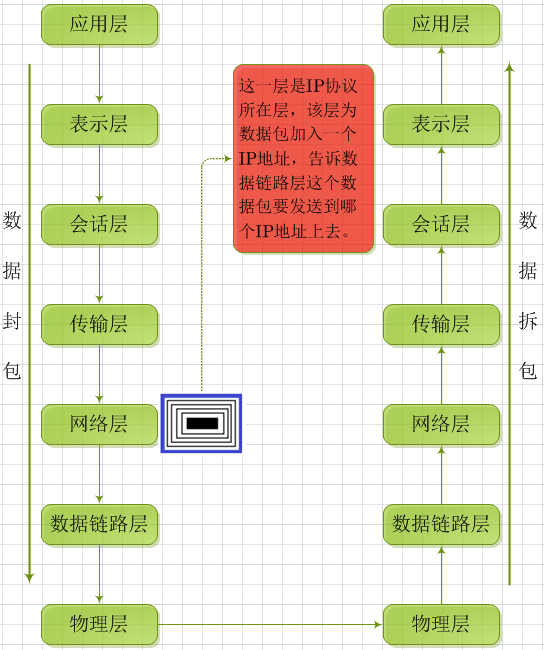


传输层

TCP协议和UDP协议所在的层次。到了这一层，已经将数据该有的信息打包完成，紧接着通过TCP或者UDP的方式向下一层发送，同理数据会被封装入TCP或UDP的信息。

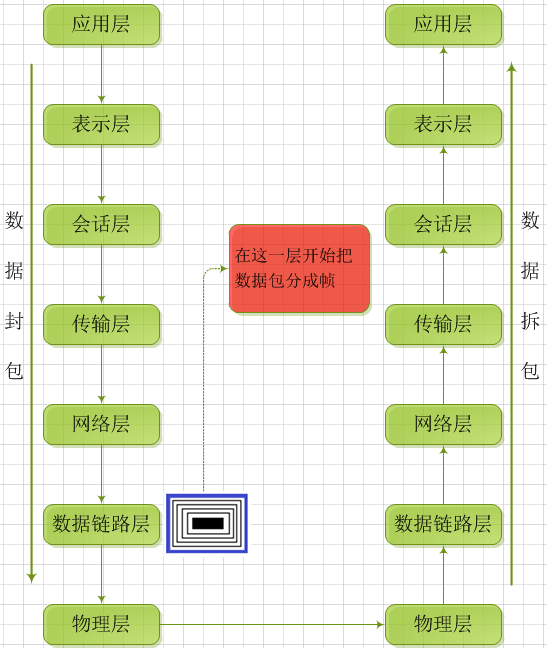


网络层 数据封包完成



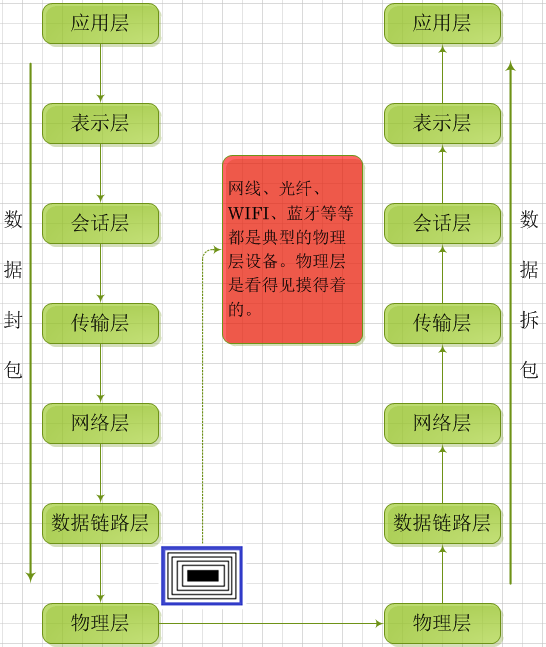
数据链路层的数据在这一层**被转换为数据包**，然后通过路径选择、分段组合、顺序、进/出路由等控制，将信息从一个网络设备传送到另一个网络设备。

数据链路层 数据包分帧



数据链路层的具体工作是将来自上层的数据包，拆为数据帧转发到物理层；同样，也将来自物理层的位流形式的数据，并封装成帧，传送到上一层；并且，还负责处理接收端发回的确认帧的信息，以便提供可靠的数据传输。

物理层



开始发送数据帧。

14.2 通信协议

即通讯规则，一种数据组织规则。找到了IP地址，定位到了程序A的端口号，也把消息(数据包；数据包中包含了真正要发送给程序的字符串、数字等)发送到了端口上，那么程序A需要以一种通信协议来拆开数据包；这种拆包的规则就是所谓的通讯协议。比如TCP/IP协议、Http协议等等。TCP/IP协议是国际标准化组织定义的通用协议。

14.2.1 Socket

14.2.2 UDP协议及应用介绍 TODO 2309

特点如下：

1. 将数据、源和目的封装成数据包，无需建立连接
2. 每个数据包的大小限制在64K内
3. 因为无连接，所以是不可靠协议
4. 不需要建立连接，速度快

典型应用场景：

1. 视频聊天
2. 飞秋

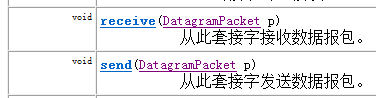
UDP协议就像发短信，TCP协议就像打电话。

* Ox01 UDP传输方式对应的对象

1. DatagramSocket.java

此类表示用来发送和接收数据报包的套接字。**事实上这个类所做的就是创建多个端点**(一个消息接收端点，n个消息发送端点)，然后在配合DatagramPacket对象来收发数据包。

这个类既能发送又能接收，两个端点进行UDP通信的话，每个端点都要有这个类。他的两个发送接收方法如下：



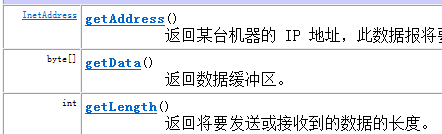
注意：这个类是阻塞式的，所以while(true)并不会陷入死循环，而且接收端往往会以这种循环方式一直等待发送端发送消息。

1. DatagramPacket.java

此类表示UDP数据报包对象。他提供了6个构造方法，最常用的2个：

DatagramPacket(**byte[]** buf, **int** length) 用来接收长度为 length 的数据包。这个构造方法用于接收数据包，**定义接收端**使用。

DatagramPacket(**byte[]** buf, **int** length, **InetAddress** address, **int** port) 构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号。这个构造方法用于发送数据包，**定义发送端**使用。当作为接收端来使用，他常用的方法摘录如下：





* Ox02 简单示例代码

**import** java.io.IOException;

**import** java.net.DatagramPacket;

**import** java.net.DatagramSocket;

**import** java.net.InetAddress;

/\*\*

\* Udp消息发送端

\* **@author** Yangcl

\*/

**public** **class** UdpSendDemo {

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

// 1 创建UDP服务，通过DatagramSocket对象

DatagramSocket ds = **new** DatagramSocket(8888);

// 2 确定发送数据内容，并封装成数据包。

// DatagramPacket(byte[] buf, int length, InetAddress address, int port)

**byte**[] bufDate = "Hello world! udp".getBytes();

InetAddress inet = InetAddress.*getByName*("172.16.3.199");

DatagramPacket dp = **new** DatagramPacket(

bufDate, bufDate.length, inet, 8089);

// 3 通过socket服务将已有数据包发送出去，send方法

ds.send(dp);

// 4 关闭资源

ds.close();

}

}

**import** java.io.IOException;

**import** java.net.DatagramSocket;

**import** java.net.DatagramPacket;

/\*\*

\* Udp消息接收端

\* **@author** Yangcl

\*/

**public** **class** UdpReceive {

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

// 1 创建udp socket接收端，建立端点 DatagramSocket(int port)

// 创建数据报套接字并将其绑定到本地主机上的指定端口，

// 方便于哪个应用程序进行处理。否则系统会随机分配一个端口

DatagramSocket ds = **new** DatagramSocket(8089);

**while**(**true**){ // 阻塞的，接收到就执行，没接收到就等待

// 2 定义数据包用于存储数据

**byte**[] buf = **new** **byte**[1024]; //udp最大64K

DatagramPacket dp = **new** DatagramPacket(buf, buf.length);

// 3 通过服接收端的receive方法将收到的数据存入数据包中

ds.receive(dp);

// 4 通过DatagramPacket对象的方法获取数据包中的内容。

String ip = dp.getAddress().getHostAddress();

String msg = **new** String(dp.getData() , 0 , dp.getLength());

**int** port = dp.getPort();

System.**out**.println("ip = " + ip + " msg = " +

msg + " 发送者端口 = " + port);

}

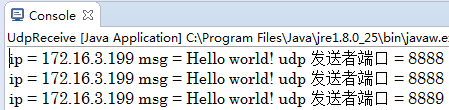
// 关闭资源

// ds.close();

}

}

结果如下：



14.2.3 TCP协议及应用介绍

特点如下：

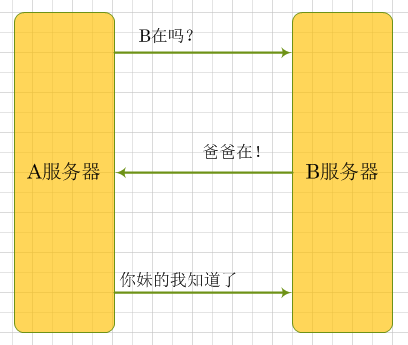
1. 必须建立连接，形成传输数据的通道
2. 连接中可以进行大数据量传输，不受限制
3. 通过三次握手完成连接，是可靠的协议
4. 由于必须建立连接，所以效率稍微低一些

典型应用场景：

1. 迅雷下载
2. 文件上传

Ox01 三次握手分析

TCP协议形成的是一个双向连接的稳定通路，之所以进行三次握手是因为：A发过去B收到了，B发过来A收没收到B是不知道的，所以为了让B知道A收到了B发送的东西，A又向B发送了一次，告诉B你发送的数据我收到了。故握手才是三次。



15 Java反射

16 注解原理

17 JVM内存模型

代码容器样式

|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| 方法名 | 功能描述 |
|  |  |
|  |  |