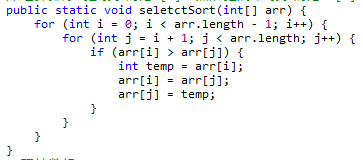
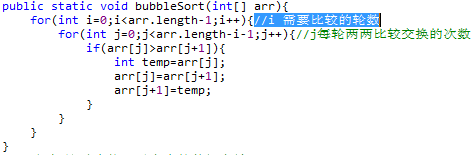
**几个基础算法**

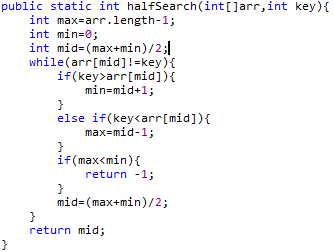
**选择:**



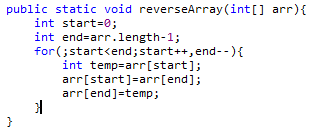
**冒泡:**



**二分：**

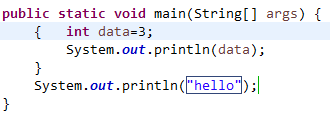


**翻转:**



**代码块**

**局部代码块**



在程序执行中顺序不变，只是在执行完成后变量data消失。减少变量的生命周期

**构造代码块**

构造函数： 构造函数的作用是给对应的对象进行初始化。

构造代码块 ： 构造代码块的作用是给所有的对象进行统一初始化,将所有构造方法中公共的信息进行抽取。*如果有多个构造方法，可以把通用的步骤写在构造代码块里。*

构造代码块的实现方式:

{

构造代码块

}

构造代码块比构造函数先执行

构造代码块要注意细节：

1. 构造函数 的代码是位于成员变量的显式初始化、构造代码块之后执行的。（不管构造函数代码的位置如何摆放）

2 . 成员变量的定义语句经过java编译器编译之后，声明变量 的语句会被提前。

3. 其实构造代码块的代码是在构造函数里面执行的。（重点）

4. 构造代码块的代码与成员 变量的显式初始化动作的先后顺序是根据当前的代码顺序而决定 的。

javap -c -l -private

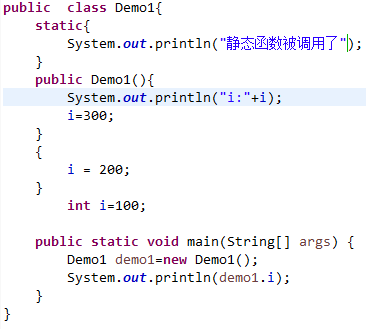
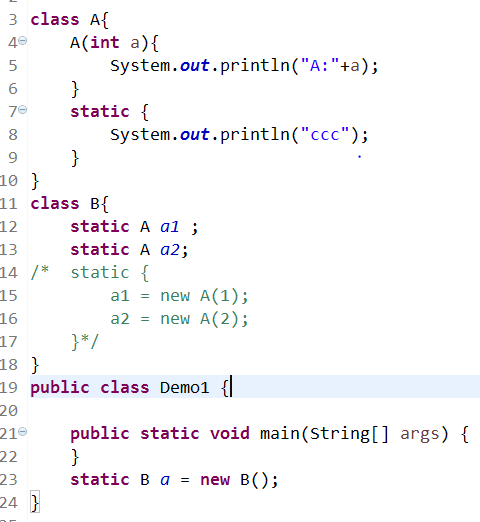


image7.png

**静态代码块**

静态代码块: 静态代码块的代码是在class文件加载的时候调用的，由于只加载一次，所以就算创建多个对象也只调用一次。优先于main函数执行

必须要使用类的时候才会加载class文件走静态代码块，如果只是声明引用则不会加载。



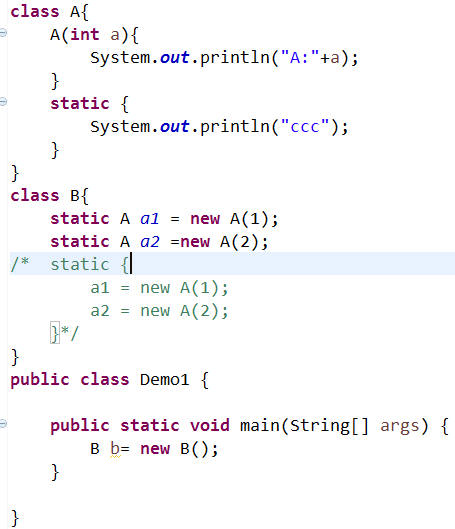
此时只声明了 A a1 Aa2没有任何输出

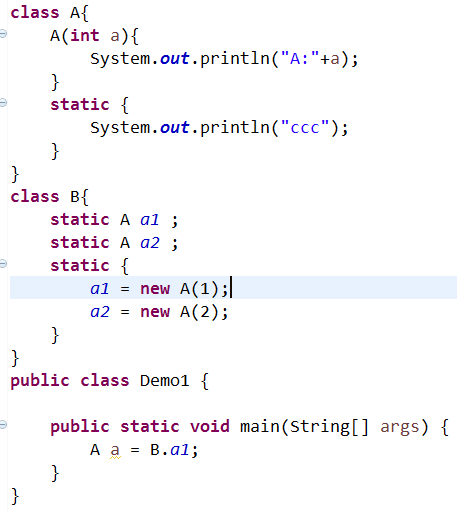
如果打开注释则输出：

ccc

A:1

A:2





上面2图都输出

ccc

A:1

A:2

**访问权限**

*访问权限   类   包  子类  其他包*

*public     ∨   ∨   ∨     ∨*

*protect    ∨   ∨   ∨     ×*

*default    ∨   ∨   ×     ×*

*private    ∨   ×   ×     ×*

注意： 在不同包中只有public、 protected两个 访问权限修饰符修饰 的内容才可以访问，而且protected必须要在继承关系下才可以访问。

**重写与重载**

重载：在同一个类中，有一个以上函数的名字相同，只要函数的参数列表或参数类型不一样即可，与返回值无关， 这些统称为方法的重载。

重写:在继承中，子类可以定义和父类相同的名称且参数列表一致的函数，将这种函数

称之为函数的重写.

重写注意:

1子类的访问权限修饰符必须大于或者等于父类的权限修饰符。

（父类方法是**protected修饰，则子类重写可以用protected或者public** private的方法无法继承，不存在重写）

2子类抛出的异常类型必须要小于或者 等于父类抛出的异常类型。

(父类方法抛出NullPointerException子类不能抛Exception)

3. 子类中的方法与父类中的方法有相同的返回类型，相同的方法名称，相同的参数列表

**修饰符**

**final:**

1. final修饰基本数据类型变量时， 该变量的值不能被修改,如果有static 则在定义 时要赋值.如果没有static,则只能在定义时或者构造中赋值，且只有一次不能修改

2. final修饰引用类型变量时，该变量不能重新指向新的对象。

3. final修饰一个方法时，该方法不能被子类重写,但可以被子类调用

4. final修饰一个类的时候，该类不能被继承。

**abstract**

**final**与abstract不能共存:

final:它的作用 修饰类代表不可以继承 修饰方法不可重写

abstract修饰类就是用来被继承的，修饰方法就是用来被重写的。

**static** static修饰的方法可以用类名调用，

对于abstract修饰的方法没有具体的方法实现，所有不能直接调用，

也就是说不可以与static共存。

**private**

private修饰的只能在本类中使用,abstract使用来继承的所以不能同时使用

**static**

变量：

静态的成员变量是随着class文件的加载而存在，随着class文件卸载消失。

非静态的成员变量随着对象创建而存在，随着对象被垃圾回收器回收的时候就消失。

类：

除了内部类外，类一般不能拥有静态属性，构造方法不能是静态。类本身可看做一种对象，简称为类对象，类的静态成员（静态成员域和静态方法）隶属于该对象。

**继承**

1. 父类私有的成员和构造方法是不能被继承的。

2. 创建子类对象时默认会先调用父类无参的构造方法。

3, 一般给父类添加一个无参构造器，便于继承,父类若没有不带参数的构造器，子类构造器必须显式调用(用super调用)超类其他的构造器,因为必须初始化父类.

如果一个类没有显示的写上构造函数，那么系统会自动加一个空参数构造函数，但写了的话，则系统不会默认加上.

**多态**

定义：父类的引用类型变量指向了子类 的对象或者是接口的引用类型变量指向了接口实现类的对象。

多态前提： 继承或者是实现关系。

多态要注意的细节：

1. 多态情况下，子父类存在着同名的成员变量时，默认访问的是父类的成员变量。

2. 多态情况下，子父类存在着同名的静态函数时， 默认访问的是父类的成员函数。

3. 多态情况下，子父类存在着同名的非静态函数时，默认访问的是子类的成员函数。

4. 多态情况下，不能调用子类特有的方法或者不能访问 子类特有的成员变量.

如下面匿名内部类第二种写法是多态的表现

总结： 多态情况下，子父类存在着同名的成员时，默认都是访问父类的成员，除了同名的非静态函数是访问子类的成员。

**抽象类**

1：有抽象函数的类，该类一定是抽象类。

2：抽象类中不一定要有抽象函数。

3：抽象类不能使用new创建对象

创建对象，使用对象的功能，抽象类的方法，没有方法体。

4：抽象类主要为了提高代码的复用性，让子类继承来使用。

5：编译器强制子类实现抽象类父类的未实现的方法。

可以不实现，前提是子类的也要声明为抽象的。

6:抽象类中一定有构造函数。主要为了初始化抽象类中的属性。

**接口**

接口的出现是为了实现多重继承的特性

1接口中的所有属性 默认的修饰符是 public static final即静态常量,既然是常量，那么定义的时候必须赋值。

2接口中的所有方法 默认的修饰符是 public abstract，不能是final protect和private

3类实现接口可以通过implements实现，实现接口的时候必须把接口中的所有方法实现,一 个类可以实现多个接口,也可以不实现，但要表示为抽象类.

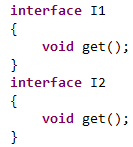
4如果实现类中要访问接口中的成员，不能使用super关键字。因为两者之间没有显示的继承关系，况且接口中的成员成员属性是静态的。可以使用接口名直接访问。

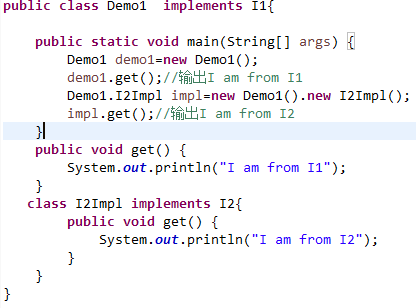
5接口与接口之间的关系是继承,一个接口可以继承多个接口

6若一个接口的方法和父类的方法相同，那么一个类同时继承父类又实现接口时，那么无法接口的方法，默认使用父类的方法.

**Java中一个类要怎么实现两个接口中相同的方法名?**

内部类实现



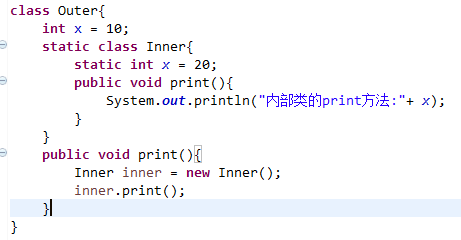


如果上面Demo1同时**implements** I1,I2，则只会实现一次get方法.

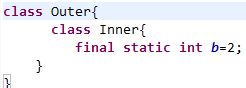
**内部类**

**成员内部类**

在A类的成员域位置定义一个B类



如果内部类的成员用static修饰 则该类也要用static，除非用final static 再赋初始值



内部类在其他类的创建方式

如果内部类没有被static修饰，则创建方式为

Outer.Inner inner = new Outer().new Inner();

如果内部类被static修饰，则创建方式为

Inner inner=new Inner();

如果被private 修饰则在其他类无法创建Inner,但可以在外部类提供方法,执行内部类的方法

new Outer().print();

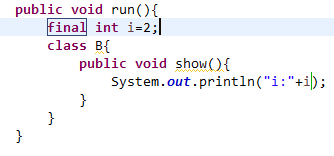
**局部内部类**

在A类的方法内部定义另外一个B类

局部内部类访问一个局部变量的时候，局部变量必须使用final修饰，

当run方法执行完毕后，i变量就从内存中消失，但是class B的实力对象还没消失，

show方法还访问着i变量，所以得加上final延长i的生命周期

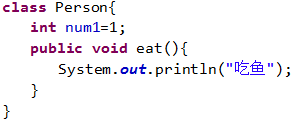


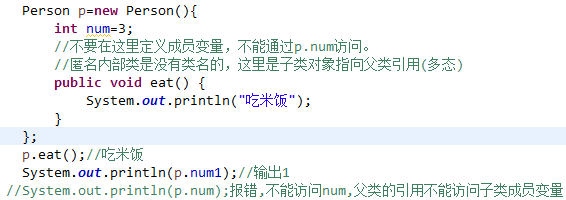
**匿名内部类**

两种书写方式:

1 new 父类或者接口(){ 执行代码….};类似new Thread().start();

2:





**String**

创建字符串一共四种方式

字符串直接量方式：

String str=”1234”;

new String()方式：

String str=new String(“1234”);

valueOf方式:

String str=String.*valueOf*("1234");

连接方式:

String str=”12”+”34”;

字符串池:

存放字符串直接量，“+”号连接以及intern创建的对象.具有相同序字符列的字符串只对应一个字符串实例对象.

intern:雍俊海p139

s1.intern()==s2.intern() 可判断字符序列是否相同.相当于equals

**泛型**

**泛型方法**

修饰符 <T> 返回值类型 函数名(形式参**数){ //方法的括号里可以使用T 作为一个类型**

**}**

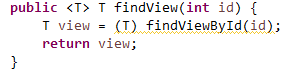
泛型方法要注意的细节：

1. 在方法上自定义了泛型，那么方法上自定义泛型的具体数据类型是在传递实际参数 的时候确定的。

2. 自定义泛型使用的标识符只需要符合标识符的命名规范即可, 但是一般我们都是使用一个大写T字母表示。

有返回值的泛型:

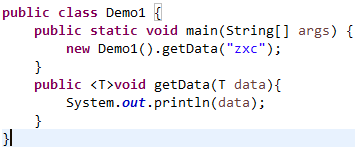
<T> 声明了泛型T，则方法内部和形式参数可以使用泛型T，此时返回值类型也为T；



传递参数时候要求返回TextView，则findView 左边的<T>为TextView，其他T和返回值类型T也会变成TextView。

image19.png

无返回值的泛型:



**泛型类**

class 类名<声明自定义的泛型>{

}

泛型类要注意 的细节：

1. 在类上自定义泛型的具体数据类型是在使用该类创建对象的时候指定的。

2. 如果一个类已经自定义了泛型，在使用该类创建对象 的时候没有指定泛型的具体数据类型，那么默认的数据类型为 Object.

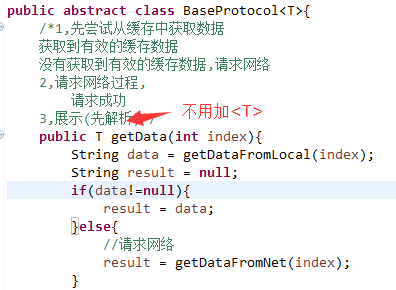
3. 静态方法不准使用类上自定义的泛型, 如果需要使用则要在方法上自定义,非静态方法则用的是类上定义的泛型



下面创建了泛型为Integer的MyArrays对象，所以reverse方法的输入参数为integer[] 数组

image22.png

泛型类定义的T ，在整个类除了静态方法之外都可以使用，所以非静态方法不用再声明<T>



**泛型接口**

interface 接口名<T>{

}

泛型接口要注意的细节：

1. 泛型接口上的自定义泛型的具体数据类型是在实现该类的时候确定。

2. 泛型接口上自定义的泛型如果在实现接口的时候没有指定具体的数据类型，那么默认为Object数据类型。

需求： 泛型接口上的自定义泛型实在实现该接口的时候指定 的，目前我想延迟指定泛型接口上的自定义泛型的具体数据类型，想在创建接口实现类对象的时候再指定。

格式：

class 类名<T> implements 接口名<T>{

}

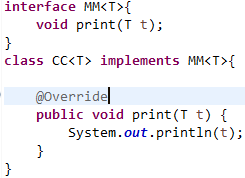


image25.png

**泛型上下限**

作用在方法上

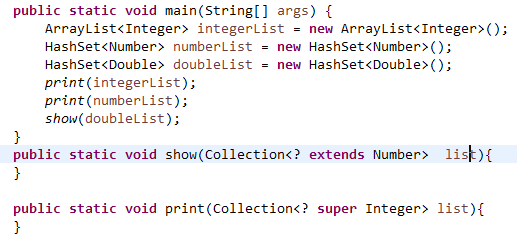
"？" 是泛型里面的通配符。

？ super Integer 泛型的下限 适用于Integer或者是Integer父类类型的数据。

？ extends Number 泛型的上限 适用于Number或者是number的子类类型数据。

需求1： 定义一个方法可以接收任意类型 的集合对象，接收的集合对象只能存储Integer或者是Integer的父类数据。

需求2： 定义一个方法可以接收任意类型 的集合对象，接收的集合对象只能存储Number或者是Number的子类数据。



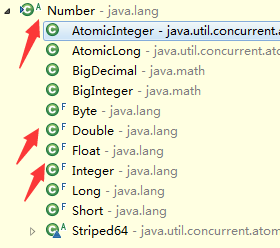
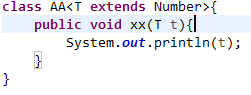


image28.png

由于Double不是Integer父类，所以报错

作用在类上



class a<T> ==class a<T extends Object>

**枚举**

枚举类要注意的细节：

1. 枚举类也是一个特殊的类。

2. 枚举值的默认修饰符 是 public static final.不能是protected以下

3. 枚举值是就是所属枚举类的类型。

4. 枚举类的构造方法必须是私有化的。

5. 枚举值必须位于枚举类中的第一个语句。

6. 枚举类是允许存在抽象方法的， 如果枚举类存在着抽象方法，那么枚举值就必 须要实现该抽象方法。

**enum** Gender{

***man***("男人"),***woman***("女人");

**private** String info;

**private** Gender(String info){

**this**.info=info;

}

**public** String toString() {

**return** info;

}

}

**可变参数**

可变参数要注意的细节：

1. 形式参数使用了可变参数的时候，调用该方法的时候可以传参数也可以不传参 2. 可变参数实际上就是一个数组对象。

3. 可变参数必须要位于形式 参数的最后一个位置上。

4. 在一个方法中最多只能出现一个可变参数，因为最后位置只有一个。

**public** **static** **void** add( **int**... arr){

**int** sum = 0;

**for**(**int** i : arr){

sum+=i;

}

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

}

*add*(1,1,2,3); //jvm会把1,1,2,3封装到一个数组对象中， sum=7

如果传入的参数就是一个object[]数组，那么就直接使用这个数组，不会转换

**增强For循环**

引入增强for循环的原因：在JDK5以前的版本中，遍历数组或集合中的元素，需先获得数组的长度或集合的迭代器，比较麻烦！

因此JDK5中定义了一种新的语法——增强for循环，以简化此类操作。**增强for循环只能用在数组、或实现Iterable接口的集合类**上

语法格式：

|  |
| --- |
| for(变量类型 变量　：需迭代的数组或集合){} |

1.增强for循环和iterator遍历的效果是一样的，也就说

增强for循环的内部也就是调用iteratoer实现的(可以查看编译后的文件)，但是增强for循环 有些缺点，例如不能在增强循环里动态的删除集合内容。不能获取下标等。
2.ArrayList由于使用数组实现，因此下标明确，最好使用普通循环。for循环 +get(location) 方法
3.而对于 LinkedList 由于获取一个元素，要从头开始向后找，因此建议使用 增强for循环，也就是iterator。

下面是上面的完整写法

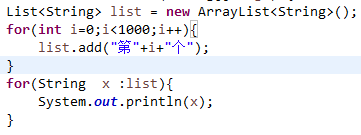


image31.png

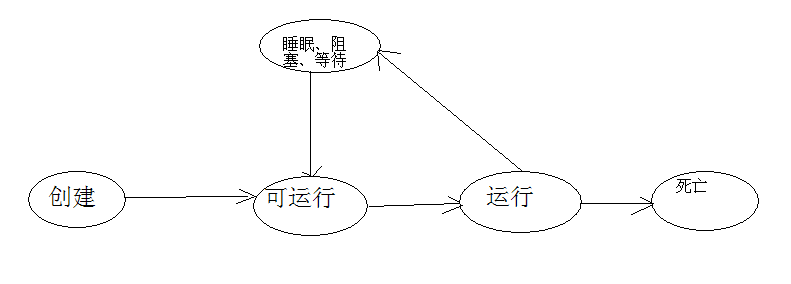
**线程**

多个线程常常因为共享内存等而需要同步处理

进程：正在运行的程序，负责了这个程序的内存空间分配，代表了内存中的执行区域。

线程：就是在一个进程中负责一个执行路径。

多线程：就是在一个进程中多个执行路径同时执行。



创建：新创建了一个线程对象。

可运行(就绪)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取cpu的执行权。

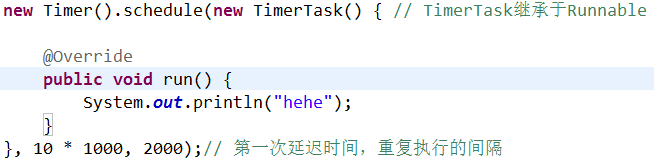
运行：就绪状态的线程获取了CPU执行权，执行程序代码。

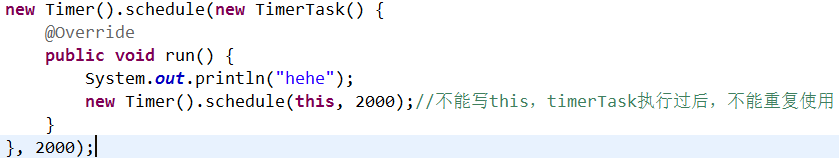
阻临时塞: 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

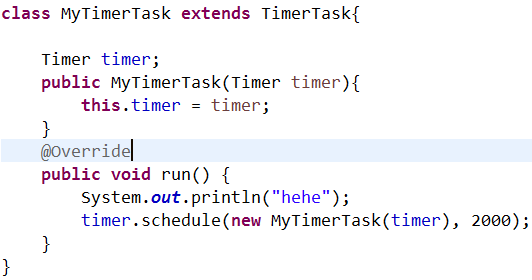
死亡：线程执行完它的任务时。

**定时器**

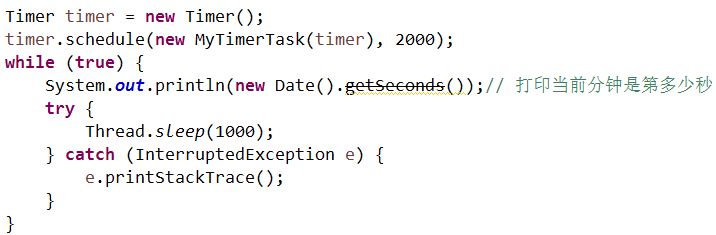
用法

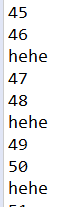






重复执行任务

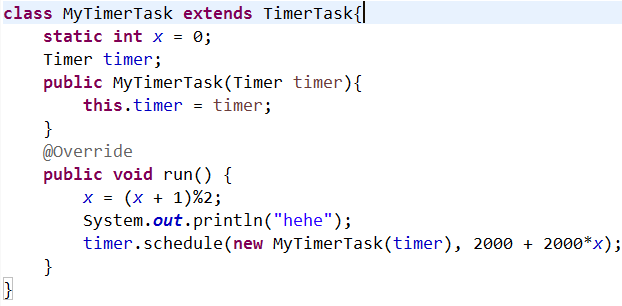




交替执行 2s 4s

1：静态变量

2：两个TimerTask，Task1中new Task2,Task2中new Task1.



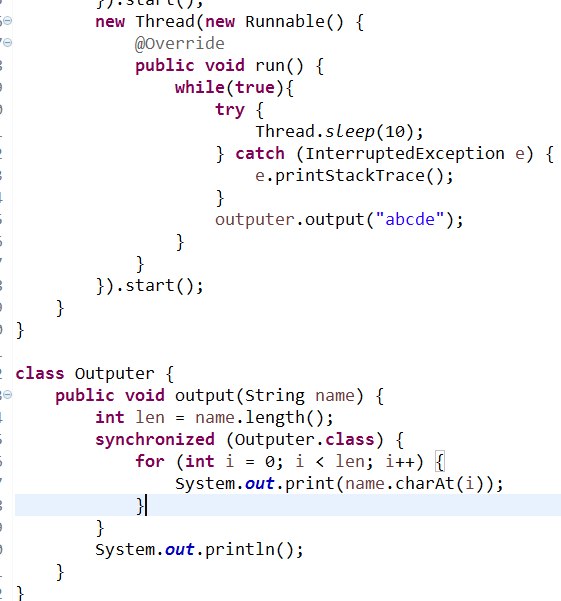
**同步代码块**

1. 任意对象都可以作为锁对象,只要锁对象相同就可保证多线程访问同步。

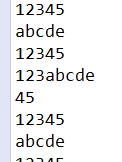
2. 锁对象一定要被多个线程共享的。

3. 一个线程调用sleep方法是不会释放锁对象的，其他线程继续等待

4. 多个线程调用同一个类的同一部分代码，可用synchronized修饰保证同步



上图多个线程使用同一个outputer对象，如果没有**synchronized**可能输出错误.



**同步函数**

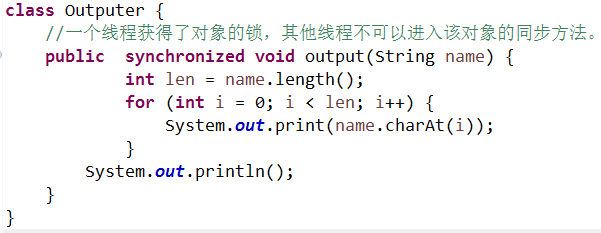
一个线程获得了对象的锁，其他线程不可以进入该对象的同步方法。

在锁释放之前，任何其他线程都不能进入同步代码（不可以进入该对象的任何同步方法）。

要用到共同数据(包括同步锁)或者共同算法的若干方法应该归在同一个类身上

如果是非静态同步函数那么当前函数的锁对象是this对象，静态函数的锁对象是当前函数所属类的字节码对象，所以静态方法和以当前类字节码为锁的代码块同步。

同步函数 的锁对象是固定的，不能修改,建议同步函数用static修饰,保证锁对象唯一



**线程通讯**

java虚拟机为每个对象配置一把锁用于线程同步和一个等候集(线程池)

线程通讯相关方法：

wait() 某个线程如果调用了wait方法，那么该线程会进入以锁对象线程池中等待, 这时会释放锁。

notify() 唤醒锁对象的线程池中等待线程的其中一个。

notifyAll() 唤醒以锁对象线程池中的所有等待线程。

进入等待状态的线程必须要由同一个锁对象调用notify才能唤醒

生产者与消费者中，这里锁对象可以是被共享资源的对象,当生产者操作自己时，等操作完毕后调用锁对象.notify()唤醒消费者线程开始操作，生产者再调用锁对象.wait()让生产者进入锁对象自己线程池中等待,然后等待消费者操作完毕后唤醒生产者

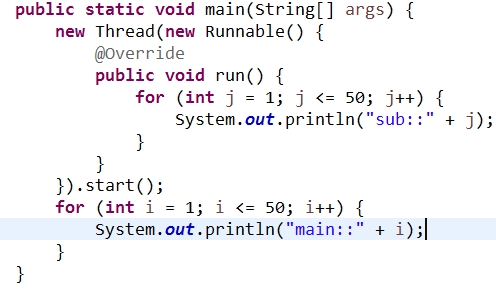
jdk1.5出现 的新锁对象-------Lock.

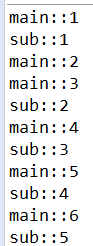
Lock 与 synchronized 的区别：

1. synchronized获取锁与释放锁的过程是隐式的， 而Lock获取锁与释放锁的过程是显式。

2. synchronized的监听器是隐式，而且只能有一个监听器。 而Lock的监听器是显式的，并且一个锁对象可以产生多个监听器.

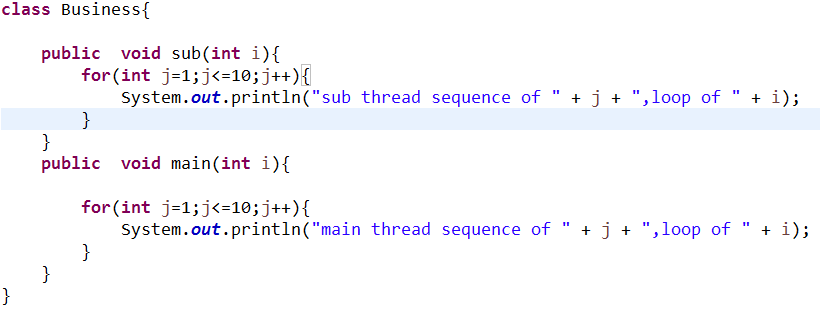
**生产者消费者**

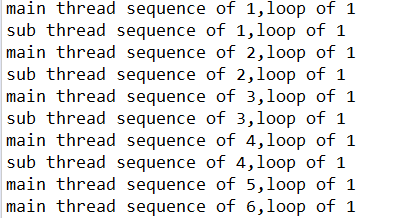




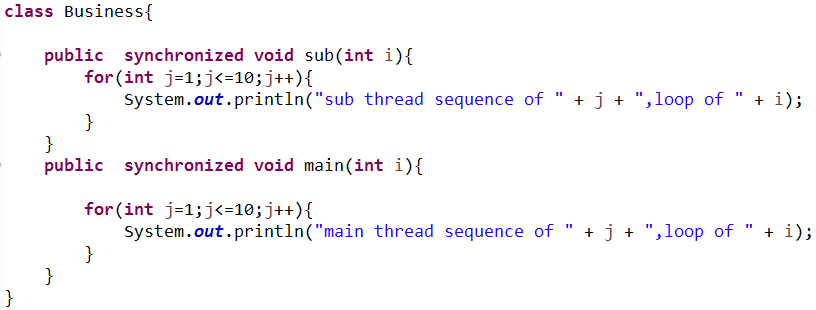
没有使用同步技术

需求子线程打印10次，再主线程打印10次，再子线程。。轮询50次

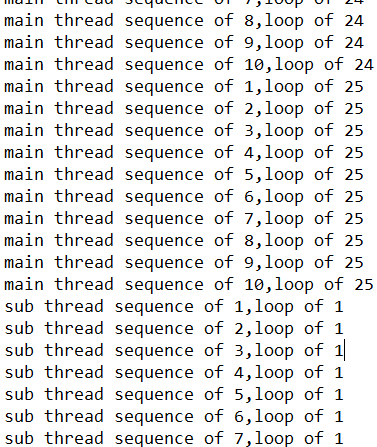




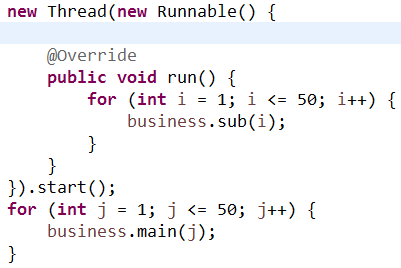
若无同步则

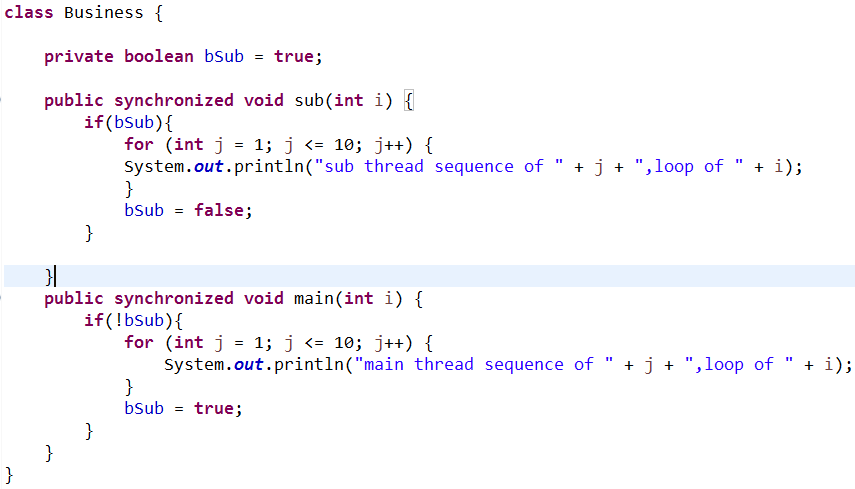


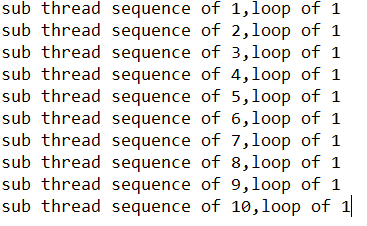
若两个方法加了sync,子主线程每一次打印不会被抢占



若没有使用wait和notify

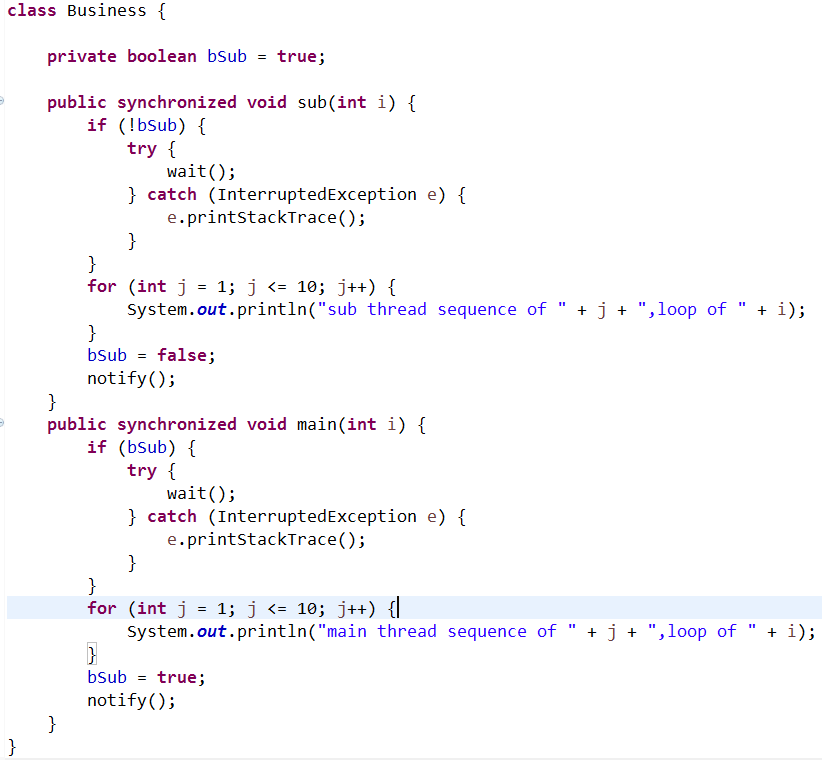






只打印一个子线程的循环，或者少量主线程循环。原因是先主线程循环50次的时候bSub一直为true，没有打印，子线程轮询50次sub方法时候，第一次改为了false。第二次就不打印了。 若主线程有阻塞，子线程改为了false，那么主线程轮询还未走完有可能打印。

加上锁的wait和notify，让线程等待和提醒

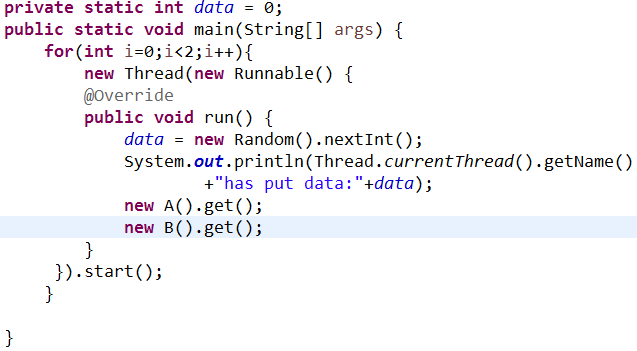


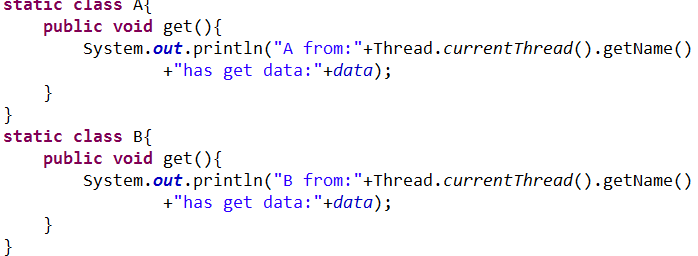
这样就实现了生产着消费者模型。这里的notify和wait是锁对象也就是这个类的对象的方法.子和主线程是在这个对象的线程池中等待. 这里的if最好换成while，多判断一次，因为可能出现伪唤醒。 比如sub正在等待，还未改为true就被伪唤醒，那么！bSub伪true，while（true）则再进入睡眠.

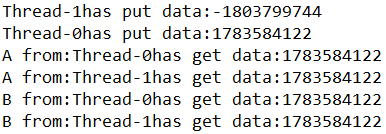
**线程范围共享变量和ThreadLocal**

某个变量在当前线程范围内是多个模块共享同步的。

若只是一个静态变量，多线程可以共同操作此变量

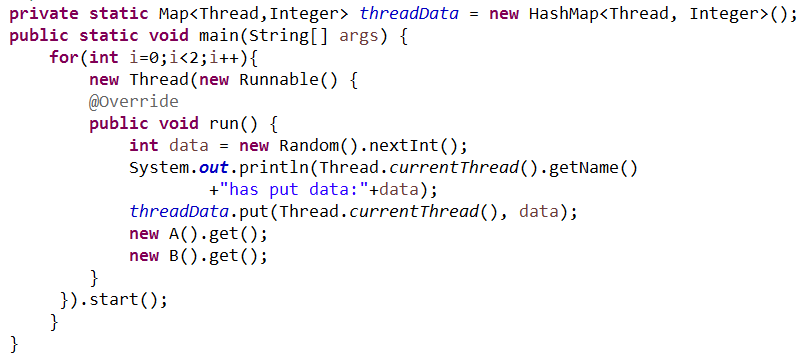


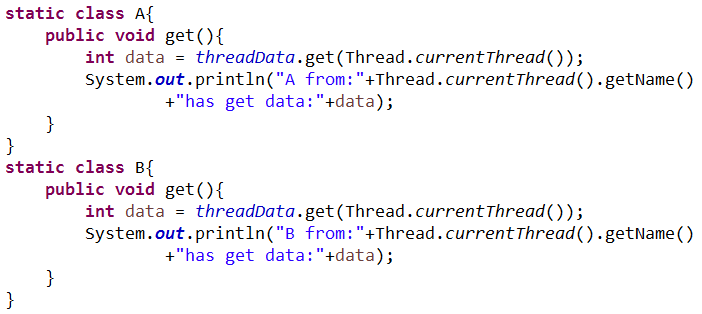


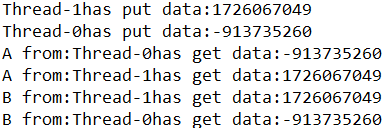


此时data 只有一份，无法做到一个线程的变量被多个模块共享，多个线程间独立

创建一个以线程我key的集合可以解决此问题

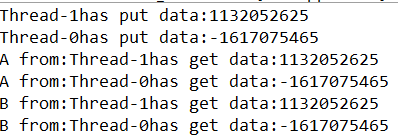






ThreadLocal





set方法设置的值与当前线程绑定了

设计线程范围共享的对象：



每次调用getInstance方法的时候回得到当前线程绑定的对象



**多个线程共享数据的方式**

例子：卖票：多个窗口同时卖这100张票，票就需要多个线程共享

a、如果每个线程执行的代码相同，可以使用同一个Runnable对象，这个对象中有共享数据。

卖票就可以这样做，每个窗口都在做卖票任务，卖的票都是同一个数据。

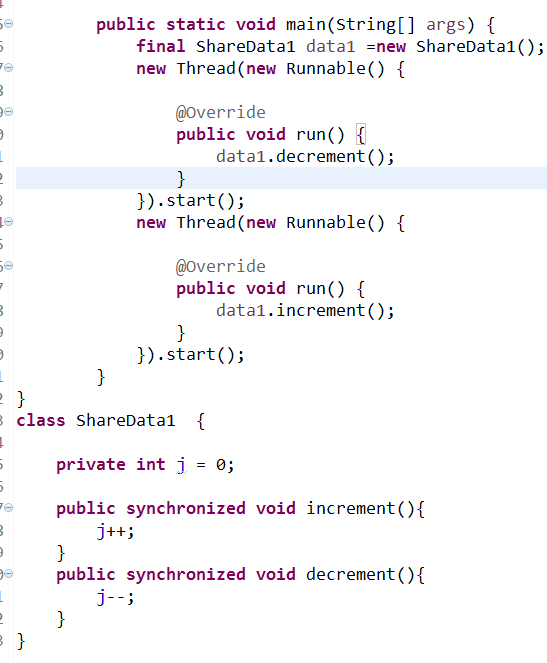
b、如果每个线程执行的代码不同，就需要使用不同的Runnable对象，有两种方式实现

Runnable对象之间的数据共享：

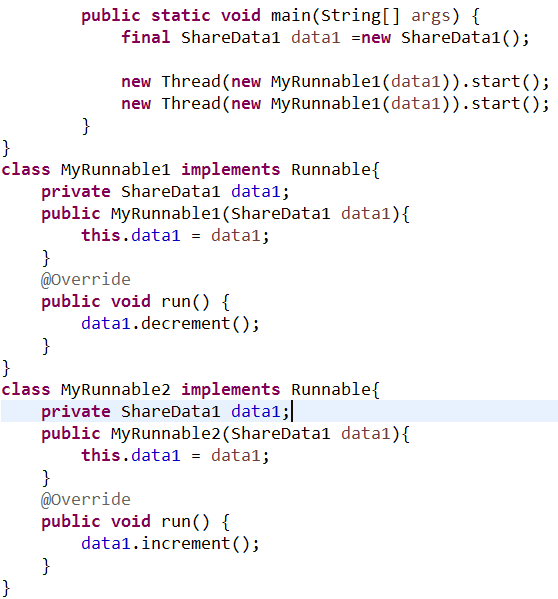
a）将共享数据单独封装到一个对象中，同时在对象中提供操作这些共享数据的方法，可以方便实现对共享数据各项操作的互斥和通信，创建不同的runnable实现类,里面维护共享数据的对象。

b）将各个Runnable对象作为某个类的内部类，共享数据作为外部类的成员变量，对共享数据的操作方法也在外部类中提供，以便实现互斥和通信，内部类的Runnable对象调用外部类中操作共享数据的方法即可。

注意：要同步互斥的几段代码最好分别放在几个独立的方法中，这些方法再放在同一个类中，这样比较容易实现它们之间的同步互斥和通信。



或者



**卖票问题**

多个线程修改同一个静态变量

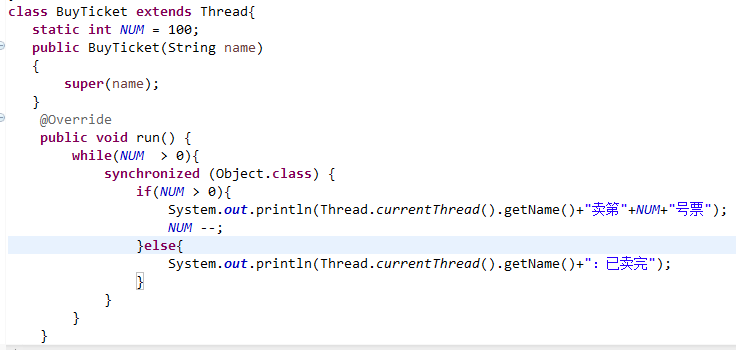
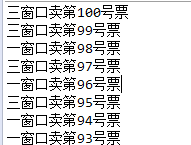
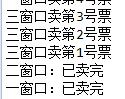
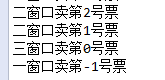


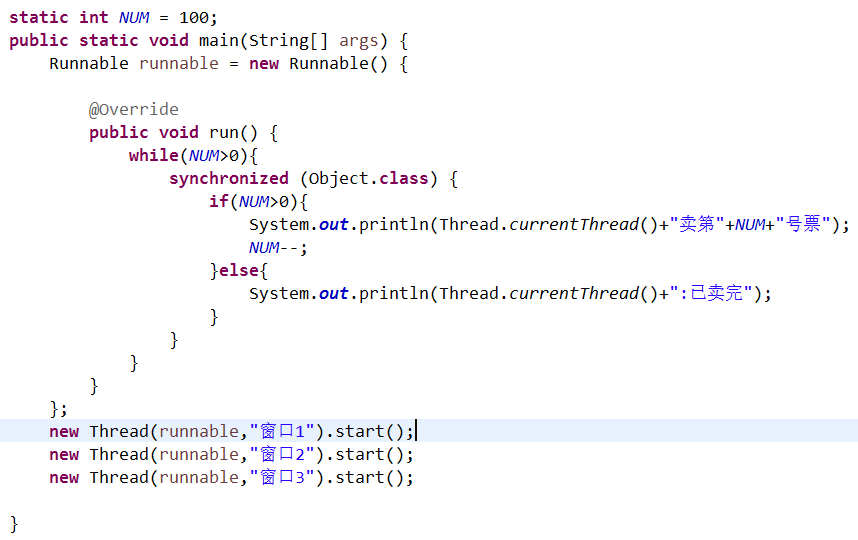
image65.png





此例子中同步代码块内还需要做一次if判定,不然会出错:





或者

**java5原子性操作类**

*一个程序,它要么完整的被执行,要么完全不执行,这种特性就叫原子性.*

java.util.concurrent包

java.util.concurrent.atomic.AtomicInteger 解决多线程操作整数的问题

AtomicIntegerArray 整数数组

AtomicIntegerFieldUpdater 操作类中的基本数据，调用newUpdater

**线程池**

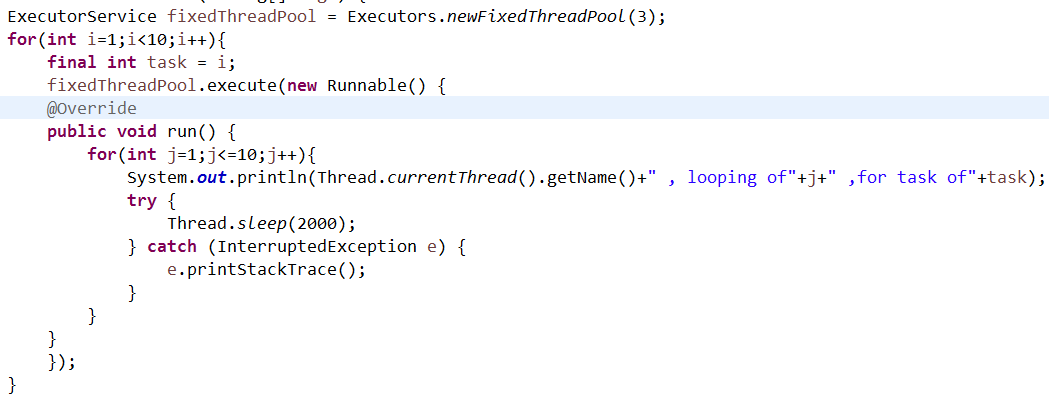
ThreadPoolExecutor

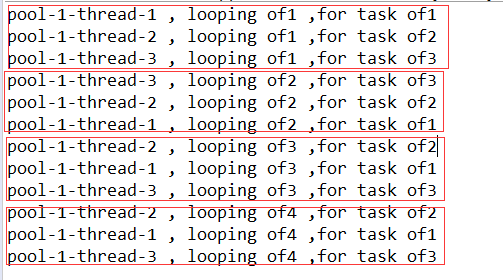
介*绍****new Thread的弊端及Java四种线程池的使用：***

http://www.trinea.cn/android/java-android-thread-pool/

***newFixedThreadPool***

***让线程池执行，10个任务，每个任务打印10次，每打印一次停2S***



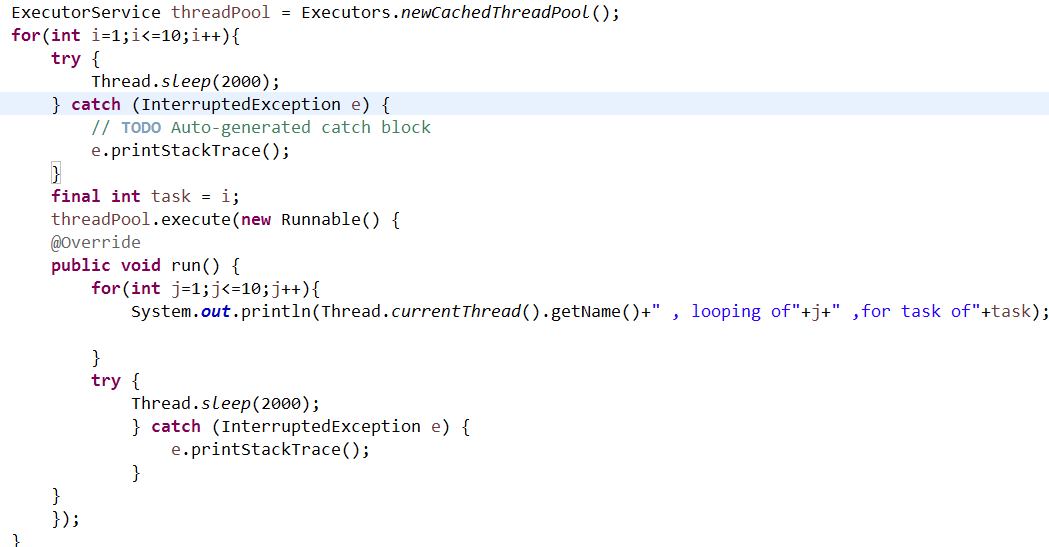


最多三个线程，同时抢占3个任务。10个任务会在最开始就提交完毕

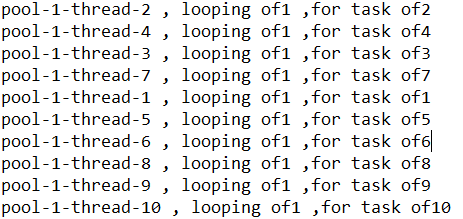
shutdown 所有任务执行完毕后销毁

shutdownNow 立即销毁，任务可能才刚开始执行

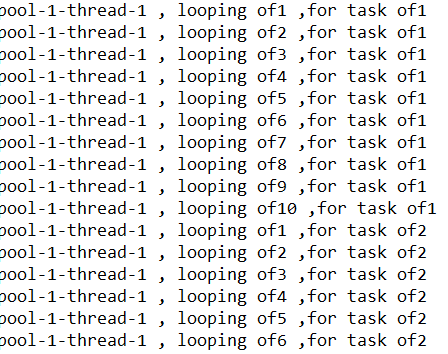
***newCachedThreadPool 缓存线程池***



如果在run方法的for下面睡眠，则会分配10个线程工作:



如果是在第一个for那里睡眠，则只有一个线程工作



如果两处都睡眠，则分配少量线程工作.

*线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。*

*第一个for那里不睡眠则，瞬间创建10个任务，所以有10个线程.*

**死锁**

java同步机制解决线程安全问题，但是同时也引发了死锁问题。

死锁现象： 多线程互相等待对方的资源。

死锁现象出现的根本原因：

1. 存在着多个线程，而且多个线程必须要共享两个或者两个以上的资源。

死锁现象的解决方案： 没有方案，只能尽量避免。（银行）

**网络**

IP Internet Protocol 互联网协议

URL Uniform Resource Locator 统一资源定位地址

网络地址两种表示法:

1：由4个整数组成的的IP地址 如 166.111.4.100

IP地址：其实IP地址的本质是由32个二进制数据的组成的， 为了方便人类去记忆，所以就把32个二进制数据分成了四段，每段8个。 0-255 00000000-00000000-00000000-00000000

2: 域名 如 www.tsinghua.edu.cn

这两个表示的是同一个网络地址

在ｊａｖａ中网络通讯又称作为Ｓｏｃｋｅｔ（插座）通讯，　通讯的两端必须要安装上Ｓｏｃｋｅｔ对象，　不同协议会产生不同的Ｓｏｃｋｅｔ对象。

**TCP**

Transmission Control Protocol 传输控制协议

Tcp特点：

1. tcp是面向连接的，是基于IO流进行数据传输，tcp的服务一旦建立，马上要与服务端连接。

2. 发送 的数据没有大小限制。

3. tcp是面向连接的，发送数据之前使用了三次握手的机制，保证数据的完整性。

4. tcp是分客户端和服务端 的。

5. tcp因为是面向连接的，所以效率低。

**udp**

User Datagram Protocal 控制网络数据传输协议

udp协议：

1.将数据封装为数据包，不需要建立连接(不需要确认通讯的对方是否在线)。

2.每个数据包大小限制在64K中

3.因为面向无连接，所以不可靠（有可能会出现数据包丢失）

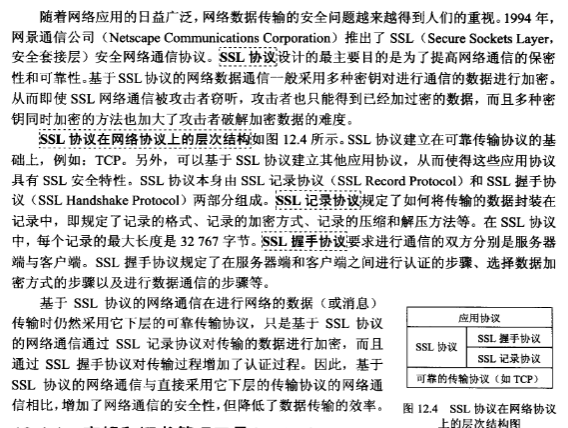
4.因为不需要建立连接，所以速度快

５.ｕｄｐ的通讯是不分客户端与服务端的，只分发送端与接收端。

比如：物管的对讲机，　飞秋的通讯，游戏　，　视频会议

**SSL**

Secure Sockets Layer 安全套接层 安全网络通信协议SSL协议





*SSL握手协议用于数据交换前的双方(客户机和服务器)身份认证以及密码算法和密钥的协商*

*SSL握手协议被用来在客户端与服务器端真正传输层数据之前建立安全机制.当客户端与服务器端第一次通信时，双方通过握手协议，在版本号、密钥交换算法、数据加密算法和Hash算法上达成一致，然后互相验证对方身份，最后使用协商好的密钥交换算法产生一个只有双方知道的秘密信息，客户端口服务器端各自根据这个秘密信息产生数据加密算法和Hash算法的参数。*

*SSL记录协议用在数据交换过程中的数据加密和数据认证，它建立在可靠的传输协议(如TCP)之上。*

*秘钥，证书和管理工具keytool*

基于SSL协议的网络通信通过密钥和证书对传输的数据进行了加密，并对通信的双方进行身份验证。 通过密钥和证书和证书的管理工具Keytool可以生成密钥库、给密钥库添加密钥项、显示密钥库信息、修改密钥库信息、删除密钥库密钥项、导出数字证书并建立信任密钥库等。

秘钥库：一系列秘钥项的集合

秘钥项：公钥public key 私钥private key 证书

**反射**

反射: 当一个class文件加载到内存的时候,jvm就会对class文件进行解剖， 解剖出这个类的所有成员信息，然后创建一个Class对象，把该类的所有成员信息都封装到该类对象中，反射就是获取到指定 的CLass文件，然后就可以操作整个类的任何成员。

就是把Java类中的各种成分映射成相应的java类,使用反射消耗资源

在反射技术中一个类 的任何成员都有相应 的类进行描述。

构造方法--》Constructor 类。

成员函数---> Method 类

成员变量--》 Field类

反射一般用于编写框架(工具类)，

**克隆**

**注解**

**初步说明**

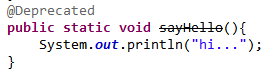
注解相当于一种标记，在程序中加了注解就等于为程序打上了某种标记，没加，则等于没有某种标记，以后，javac编译器，开发工具和其他程序可以用反射来了解你的类及各种元素上有无何种标记，看你有什么标记，就去干相应的事(比如下面sayHello方法加了@Deprecated标记，那么调用处则画上横线)。标记可以加在包，类，字段，方法，方法的参数以及局部变量上。

JDK提供的基本注解：

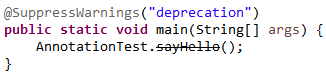
**Deprecated Override SuppressWarnings**

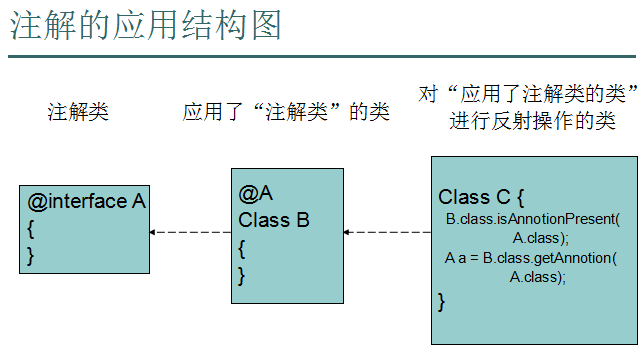
**注解作用就是向编译器传递一些信息**

标记一个方法为废弃的方法,上面用Deprecated修饰



调用者则要加上注解SuppressWarnings(“deprecation”);

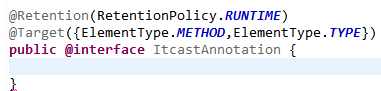




对一个类进行检查则要获得类的字节码文件

|  |  |  |
| --- | --- | --- |
| boolean | **isAnnotation**()    如果此 Class 对象表示一个注释类型则返回 true。 | boolean |
| boolean | **isAnnotationPresent**(Class<? extends Annotation> annotationClass)  如果指定类型的注释存在于此元素上，则返回 true，否则返回 false。 | Annotation |
| Annotation | **getAnnotation**(Class<A> annotationClass)           如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。 | |

**自定义注解**



Retention Target 是元注解，是用来修饰注解的注解。

Retention的值默认是RetentionPolicy.CLASS 则是将ItcastAnnotation注解保留到class文件 。 如果不设置为RUNTIME，则被ItcastAnnotation修饰的类会在运行时在内存中取消注解。

Target表示注解能应用在哪些字段上面，默认是所有，这里是能修饰方法的类（以及接口枚举等）.

**设计模式**

**工厂模式**

*工厂模式：主要用来实例化有共同接口的类，工厂模式可以动态决定应该实例化那一个类。*

http://www.cnblogs.com/zhouqiang/archive/2012/07/20/2601365.html

**观察者模式**

http://blog.csdn.net/jason0539/article/details/45055233

1:观察者观察的方法要定义抽象接口

2:被观察者接口一般要定义三个抽象方法

添加观察者,删除观察者,发布信息给观察者

3:不同的具体观察者实现抽象观察接口，被观察者实现被观察抽象接口

4:被观察实例，添加各个观察者.再根据条件通知各个观察者

**断点**

F5进入方法执行，调用了其他方法，可以进入方法一步一步执行,进入不了就本方法内下一步

F6只会在本方法内一步一步执行,本方法执行完毕后会跳出方法执行原方法

F7跳出此方法，继续执行调用这个方法的原方法

F8 运行到下一个断点处