# jvm、jre、jdk关系图

Jvm + 核心类库 == Jre Jre + 开发工具 == Jdk

# 2、跨平台性

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。

# **3、访问修饰符**



所以protected比什么都不写要更开放一点，因为它开放给子类，而default仅仅开放给同包。

# **4、super和this**



一般而言父类都要声明无参构造方法，因为子类执行构造方法时，都会默认执行super()

# java中如何跳出多重循环



# 多态

一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

父类或接口定义的引用变量可以指向子类或具体实现类的实例对象。提高了程序的拓展性。

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

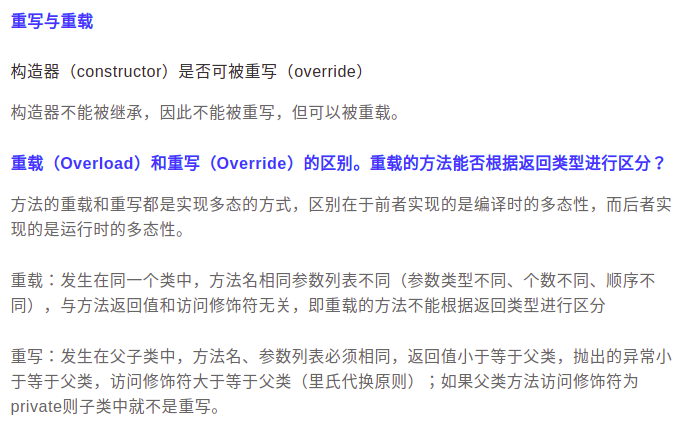
方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。

Java实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。

重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。



# **面向对象的 六原则一法则是什么？**

1. 1单一职责原则：一个类只做它该做的事情，实现高内聚原则
2. 2开闭原则：对扩展开放，对修改关闭
3. 3依赖倒转原则：面向接口编程，声明方法的参数类型、方法的返回类型、变量的 引用类型时，尽可能使用抽象类型而不用具体类型
4. 4里氏替换原则：任何时候都可以用子类型来替换掉父类型
5. 5接口隔离原则：接口要小而专，不要大而全
6. 6合成聚合复用原则：多用组合少用继承实现代码复用（将狗继承宠物改成狗类有 一个宠物成员变量）
7. 迪米特法则：低耦合原则

# 抽象类和接口的对比

抽象类是用来捕捉子类的通用特性的。接口是抽象方法的集合。从设计层面来说，抽象类是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：

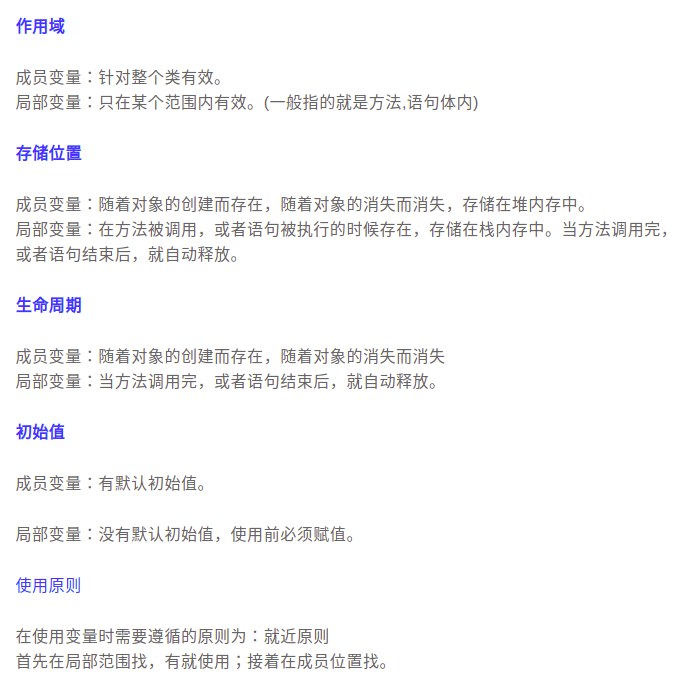
行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量少用抽象类。

选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用的功能。

# 成员变量和局部变量的对比

成员变量：方法外部，类内部定义的变量

局部变量：类的方法中的变量。



# 构造方法的特性

名字与类名相同；

没有返回值，但不能用void声明构造函数；

生成类的对象时自动执行，无需调用。

# 在一个静态方法里调用非静态成员为什么是非法的

因为静态方法为类共享，非静态成员为对象专有

# 内部类

我认为两种：

成员内部类（定义在类的成员变量位置，其中用static修饰的叫静态内部类）；

局部内部类（定义在成员方法局部变量内部，含匿名内部类(new 类名/接口)）；







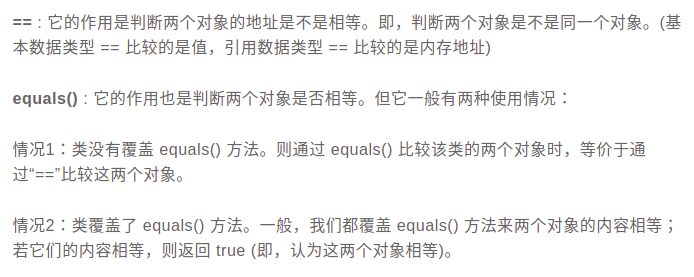


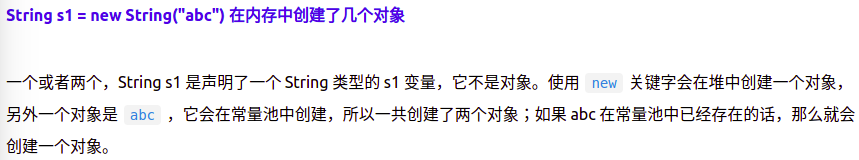






# ==和equals的区别





上述例子中有一个注释写错了，String aa = “ab”,这条语句“ab”这个实例早已经存在，从常量池返回即可，无需重新放入常量池。

String中的equals重写的思路：

1. 引用是否相等，不等，return false
2. 是否是string实例，不是，return false
3. 长度是否相等，不等，return false
4. 逐个字符比较，有一个不等，return false
5. 相等

# hashcode和equals

首先明白一点：

两个对象equals相等，那么他们的hashcode一定相等；

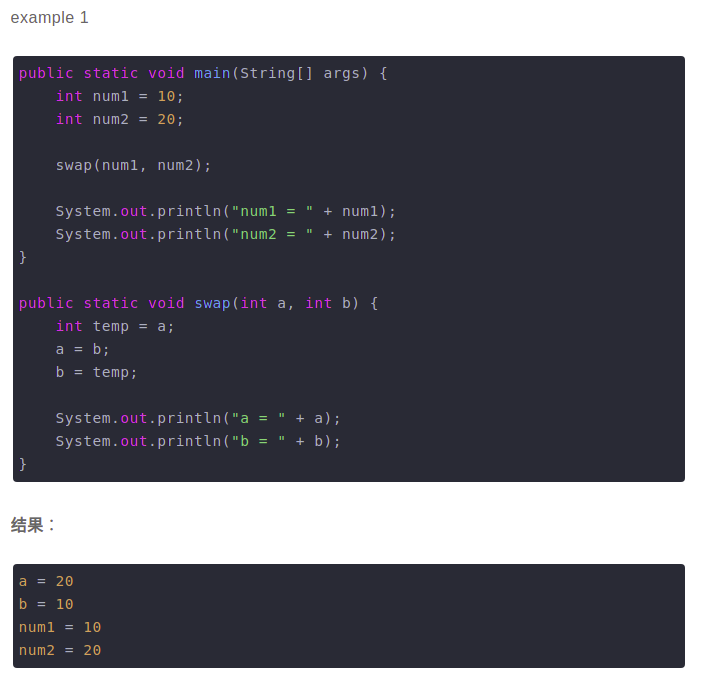
两个对象的hashcode相等，但他们的equals不一定相等；

前者证明了重写equals方法，那么一定要重写hashcode方法

后者证明了HashMap用hash值找到了对应桶，但是还是要用equals方法判断是否相等，hashcode方法减少了equals方法的调用，因为hashcode不等，那么没必要比较equals了。

# Java 中只有值传递

只有值传递，而之所以有引用传递的效果，只不过是因为你传递的值是引用的拷贝，指的还是同一个地址而已。



如上，传递的是值的拷贝



也就是说你要看方法参数是否引用类型，如果是，将改变实参的值。

但是java传递的还是值的拷贝，只不过的引用值的拷贝而已，引用值的拷贝，但是引用指的还是一个地址啊。

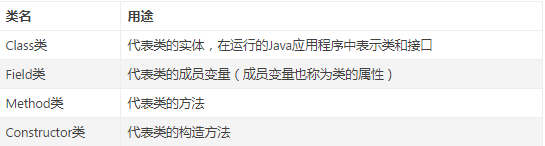
# 反射

在日常的第三方应用开发过程中，经常会遇到某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放，这时候就可以**利用Java的反射机制通过反射来获取所需的私有成员或是方法**

### 1.定义

[JAVA反射机制](https://baike.baidu.com/item/JAVA%E5%8F%8D%E5%B0%84%E6%9C%BA%E5%88%B6/6015990" \t "_blank)是在运行状态中，**对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性**；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

### 2.反射机制的相关类



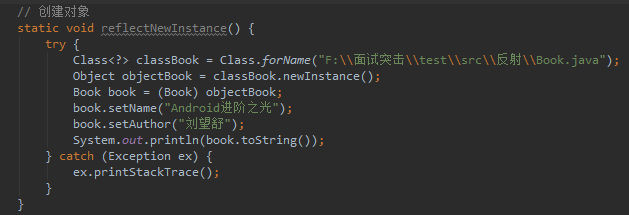
后三种类是基于class类获得的。

#### Class类



获得class类的三种方法

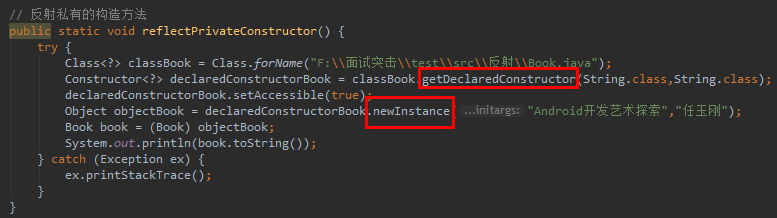


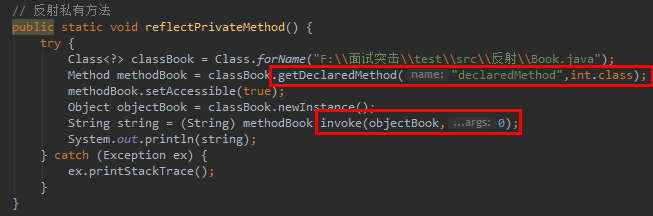














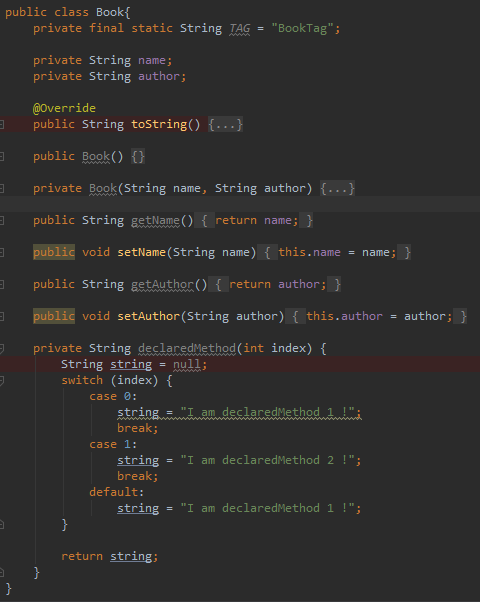
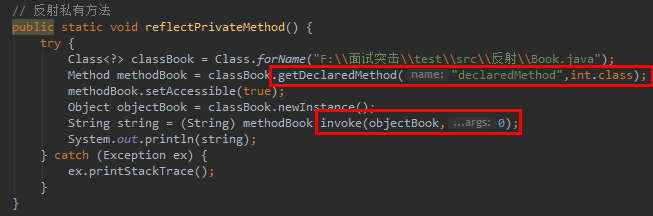
#### Field类





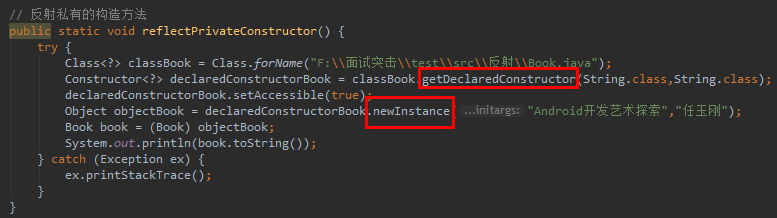
#### Method类





#### Constructor类





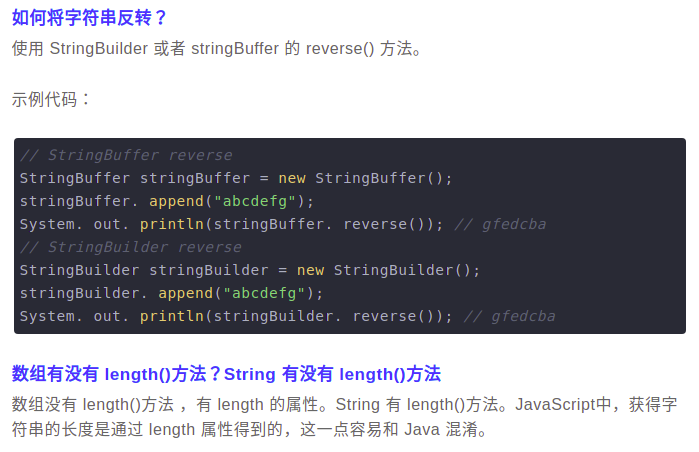
# String、StringBuffer、StringBuilder

String不是基本数据类型，只是因为觉得字符用数组太过繁琐，而设定的引用数据类型，底层仍是字符数组。



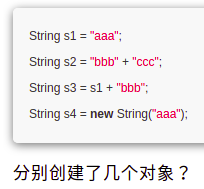






这也能说明String不可变的特点，如果可变的话，那么就会有reverse方法了，因为reverse方法就是在自身的基础上翻转元素顺序





# int和Integer



Integer对象也会有字符串常量池的概念，但是只能装内容为-128到127的对象。

## [Integer.valueOf 和 Integer.parseInt 和 new Integer区别及注意事项](https://www.cnblogs.com/hdwang/p/7009449.html)

1.System.out.println(127==127); //true , int type compare

2.System.out.println(128==128); //true , int type compare

3.System.out.println(new Integer(127) == new Integer(127)); //false, object compare

4.System.out.println(Integer.parseInt("128")==Integer.parseInt("128")); //true, int type compare

5.System.out.println(Integer.valueOf("127")==Integer.valueOf("127")); //true ,object compare, because IntegerCache return a same object

6.System.out.println(Integer.valueOf("128")==Integer.valueOf("128")); //false ,object compare, because number beyond the IntegerCache

7.System.out.println(Integer.parseInt("128")==Integer.valueOf("128")); //true , int type compare

int整型常量比较时，== 是值比较，所以1,2返回true。1，2是值比较。

new Integer() 每次构造一个新的Integer对象，所以3返回false。3是对象比较。

Integer.parseInt每次构造一个int常量，所以4返回true。4是值比较。

Integer.valueOf返回一个Integer对象，默认在-128~127之间时返回缓存中的已有对象（如果存在的话），所以5返回true，6返回false。5，6是对象比较。

第7个比较特殊，是int 和 Integer之间的比较，结果是值比较，**引用数据类型会自动拆箱，但是这句话只针对Integer和int，因为String没有基本数据类型。**

# 19、线程安全和线程不安全的单例模式？

 public class Singleton {

private Singleton(){}
// 构造方法声明为私有，所以只能生成一个对象

private static Singleton instance = new Singleton();

public static Singleton getInstance(){

return instance;

}

}

public class Singleton {

private static Singleton instance = null;

private Singleton() {}

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

if (instance == null) instance ＝ new Singleton();

return instance;

}

1. }

单例设计模式：保证类在内存中只有一个对象。

单例设计模式分为两种，一类是饿汉式，上来就new 对象，用空间换时间，二类是懒汉式，先判断是否有单例对象，如果没有再new 对象，判断需要时间，所以用的是时间换空间。

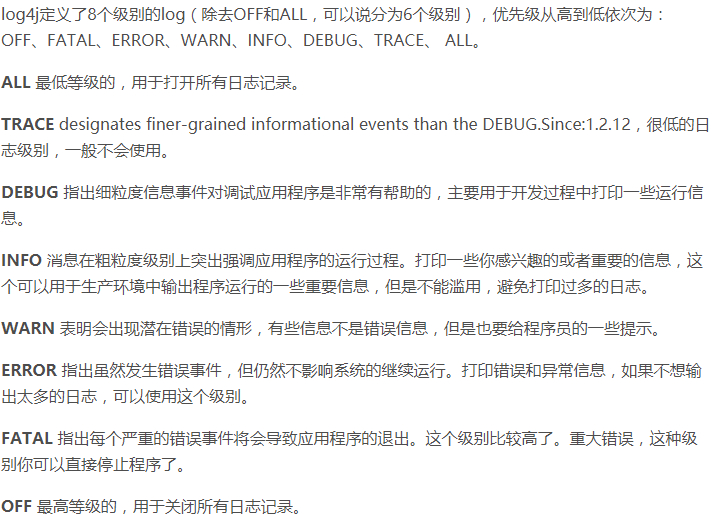
明显是饿汉式更好，但是懒汉式可能会用于面试。

在多线程访问时，饿汉式不会创建多个对象，而懒汉式可能会创建多个对象，因为饿汉式创建对象的方法私有，而懒汉式创建对象时的方法不私有，线程可能多次访问（待考证）。

在jdk手册中，如果一个类没有看到构造方法，说明该类的构造方法很有可能被私有化了。说明该类可能是单例设计模式下的类，不允许实例化对象，只允许用类名调用返回。

# 20、日志优先级

一般而言都是INFO, 可以打印警告和错误. 用到DEBUG就比较细了.



# 21、java定义二维数组的方法：

1、动态初始化

数据类型 数组名 [ ][ ] = new 数据类型[m][n]

数据类型 [ ][ ]  数组名 = new 数据类型[m][n]

数据类型 [ ]   数组名 [ ] = new 数据类型[m][n]

举例：int [ ][ ]  arr=new  int [5][3];  也可以理解为“5行3例”

2、 静态初始化

数据类型 [ ][ ]   数组名 = {{元素1,元素2....},{元素1,元素2....},{元素1,元素2....}.....};

举例：int [ ][ ]  arr={{22,15,32,20,18},{12,21,25,19,33},{14,58,34,24,66},}

# 22、正则表达式的() [] {}有不同的意思。

1. () 是为了**提取匹配的字符串**。表达式中有几个()就有几个相应的匹配字符串。

例: (\s\*)表示连续空格的字符串。(0-9) 匹配 '0-9′ 本身。

1. []是定义**匹配的字符范围**。

例:[\s\*]表示空格或者\*号。[0-9]\* 匹配数字（注意后面有 \*，可以为空）[0-9]+ 匹配数字（注意后面有 +，不可以为空）比如 [a-zA-Z0-9] 表示相应位置的字符要匹配英文字符和数字。

1. {}一般用来表示**匹配的长度**

  例: {1-9} 写法错误。比如 \s{3} 表示匹配三个空格，\s{1,3}表示匹配一到三个空格。

[0-9]{0,9} 表示长度为 0 到 9 的数字字符串

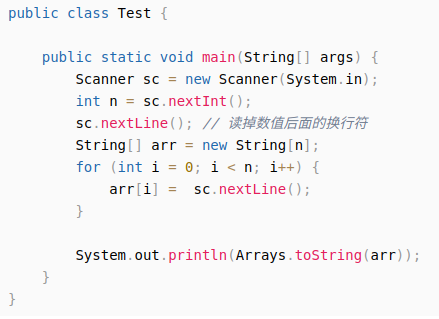
# 23、nextInt和nextLine引发的问题

nextInt：此方法**只读取整型数值**，并且在读取输入后**把光标留在本行**

next：**读取输入直到遇见空格**。此方法不能读取被空格分隔开的内容，并且在读取输入后把光标留在本行

nextLine：读取包括空格在内的输入，而且还会读取行尾的换行字符\n，读取完成后**光标被放在下一行**

如果用了nextInt后，又用nextLine时，nextLine会将nextInt后的换行符给接收，那么这样的话，接收的数组接受的数据会少了最后一个，因为前面接收了一个换行符。所以要先吧这个换行符给接收：sc.nextLine；



同理, 如果用hasNextLine作为循环跳出条件，那么循环会永远成立，跳不出循环，因为每一行后面都会有一个换行符.

# BigInteger、BigDecimal

* A:BigInteger的概述
  + 可以让超过Integer范围内的数据进行运算
* B:构造方法
  + public BigInteger(String val)
* C:成员方法
  + public BigInteger add(BigInteger val)
  + public BigInteger subtract(BigInteger val)
  + public BigInteger multiply(BigInteger val)
  + public BigInteger divide(BigInteger val)
  + public BigInteger[] divideAndRemainder(BigInteger val)
* A:BigDecimal的概述
  + 由于在运算的时候，float类型和double很容易丢失精度。
  + 所以，为了能精确的表示、计算浮点数，Java提供了BigDecimal。
  + 不可变的、任意精度的有符号十进制数。
* B:构造方法
  + public BigDecimal(String val)
* C:成员方法
  + public BigDecimal add(BigDecimal augend)
  + public BigDecimal subtract(BigDecimal subtrahend)
  + public BigDecimal multiply(BigDecimal multiplicand)
  + public BigDecimal divide(BigDecimal divisor)

# 多线程

## 进程线程间通讯原理

**1、进程是什么？**

是具有一定独立功能的程序、它是**系统进行资源分配和调度的一个独立单位**，重点在系统调度和单独的单位，也就是说进程是可以独立运行的一段程序。

**2、线程又是什么？**

线程进程的一个实体，是CPU调度和分派的基本单位，他是比进程更小的能独立运行的基本单位，线程自己**基本上不拥有系统资源**。在运行时，只是暂用一些计数器、寄存器和栈。**是程序执行的最小单位**。

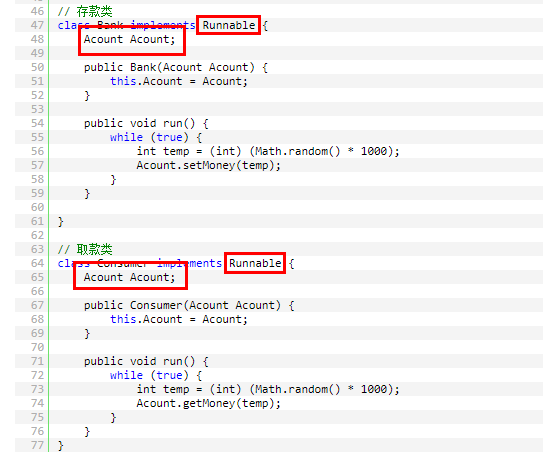
**从三个角度来剖析二者之间的区别**  
1、调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。  
2、并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行。  
3、拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

#### 1. 线程间通讯原理

##### 1、共享变量

最常见的就是共享一个对象，如银行存款取款共享一个Acount对象：





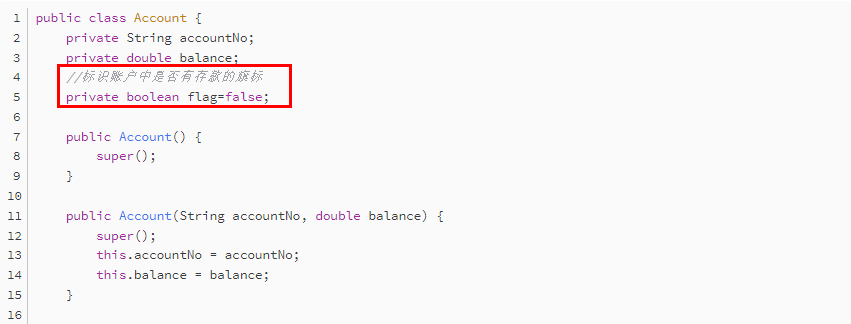
只要存的钱大于要取的钱，就一直取随机数量，每隔10秒钟存一次随机数量的钱，所以这个程序不会停止。

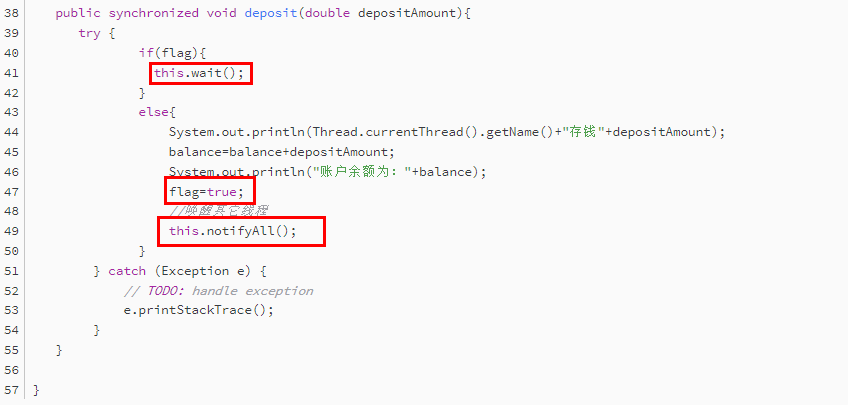


##### wait/notify机制

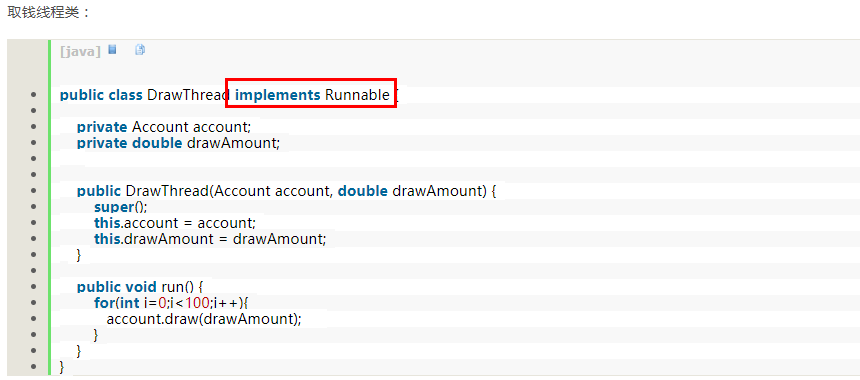
为了实现线程通信，我们可以使用Object类提供的wait()、notify()、notifyAll()三个方法，这三个方法在使用synchronized时使用：

**定义Account类**

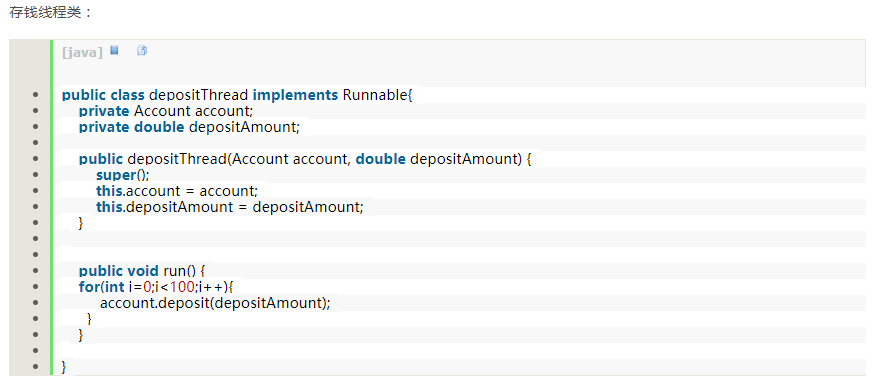




**定义取钱线程类**



**定义存钱线程类**





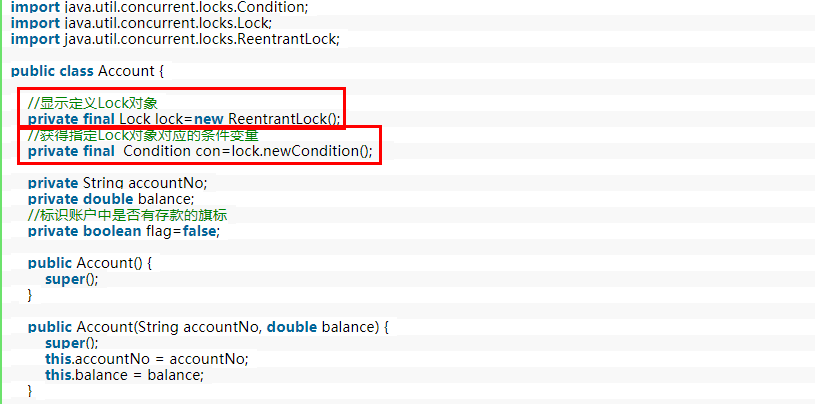
其实也用到了共享变量对象account，只是利用wait，notifyAll使其交叉运行，使得不能连续存钱和取钱。上面也用到了wait和notify，所以二者常常配合使用。

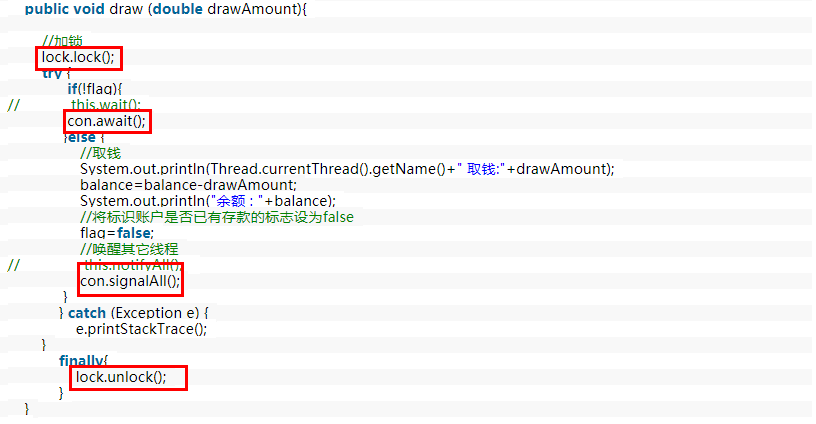


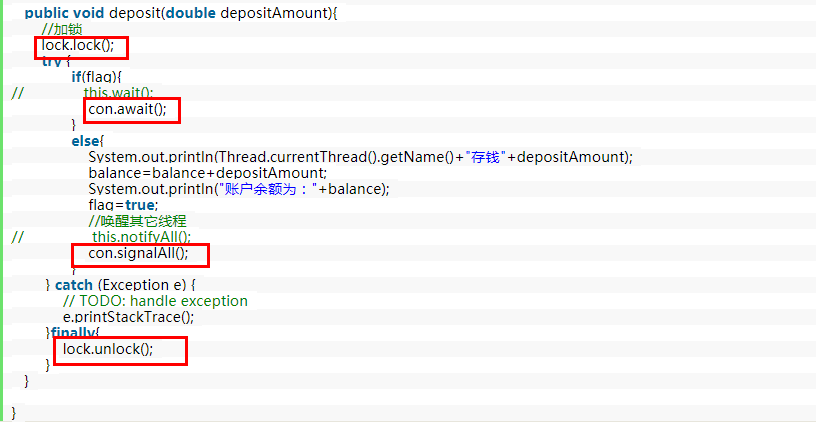
##### Lock/Condition机制

如何程序不使用synchronized关键字来保持同步，而是直接适用Lock对像来保持同步，则系统中不存在隐式的同步监视器对象，也就不能使用wait()、notify()、notifyAll()来协调线程的运行.

当使用LOCK对象保持同步时，JAVA为我们提供了Condition类来协调线程的运行。







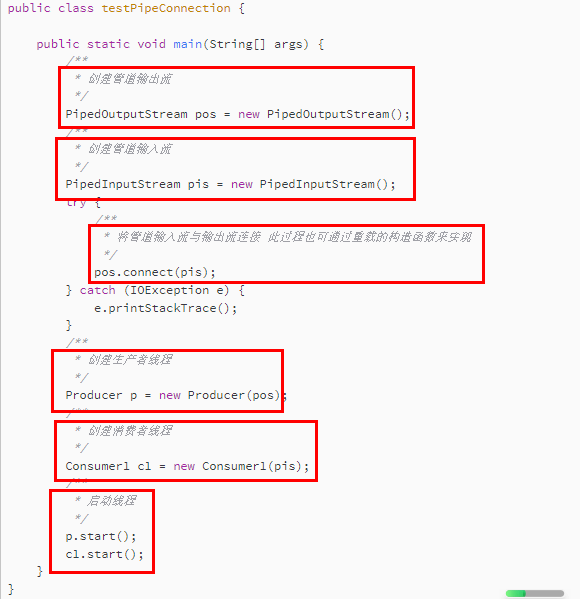
##### 管道

管道流是JAVA中线程通讯的常用方式之一，基本流程如下：

1）创建管道输出流PipedOutputStream pos和管道输入流PipedInputStream pis

2）将pos和pis匹配，pos.connect(pis);

1. 将pos赋给信息输入线程，pis赋给信息获取线程，就可以实现线程间的通讯了







输出结果：



管道流虽然使用起来方便，但是也有一些缺点

1）管道流只能在两个线程之间传递数据

线程consumer1和consumer2同时从pis中read数据，当线程producer往管道流中写入一段数据后，**每一个时刻只有一个线程能获取到数据**，并不是两个线程都能获取到producer发送来的数据，因此一个管道流只能用于两个线程间的通讯。不仅仅是管道流，其他IO方式都是一对一传输。

1. 管道流**只能实现单向发送**，如果要两个线程之间互通讯，则需要两个管道流

Java提供管道功能，实现管道通信的类有两组：PipedInputStream和PipedOutputStream或者是PipedReader和PipedWriter。管道通信主要用于不同线程间的通信。

一个PipedInputStream实例对象必须和一个PipedOutputStream实例对象进行连接而产生一个通信管道。PipedOutputStream向管道中写入数据，PipedIntputStream读取PipedOutputStream向管道中写入的数据。一个线程的PipedInputStream对象能够从另外一个线程的PipedOutputStream对象中读取数据。

#### 2.进程间通讯原理

##### （1）管道（Pipe）：

管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。一般用于线程间通讯。

**Linux管道的实现机制**  
  
从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现为：·  
 1) 限制管道的大小。实际上，管道是一个固定大小的缓冲区。在Linux中，该缓冲区的大小为1页，即4K字节，使得它的大小不象文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供write()调用写。  
 2) 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的read()调用将默认地被阻塞，等待某些数据被写入，这解决了read()调用返回文件结束的问题  
 **注意：**  
 从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。  
  
 **管道的结构  
 在Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的file结构和VFS的索引节点inode。**

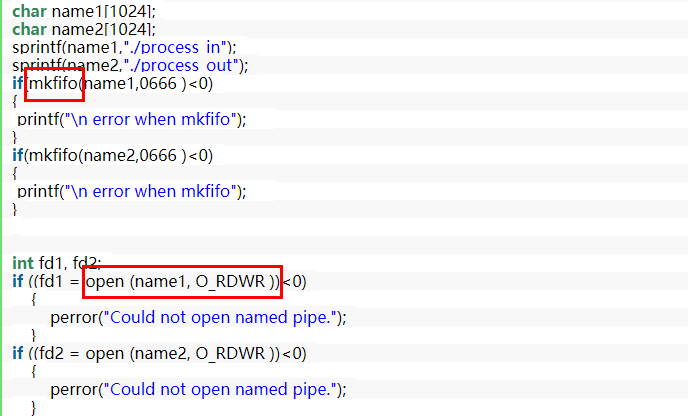
**过将两个file 结构指向同一个临时的VFS 索引节点，而这个VFS引节点又指向一个物理页面而实现的。**

##### **（2）命名管道（named pipe）：**

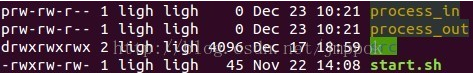
命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。

**1）首先要创建一个管道文件，这一点Java 做不到，我们要借助C/C++中的mkfifo()函数来实现。**

**以下代码创建并打开两个管道，一个用于读取输入，一个用于输出：**



**这是查看当前目录，可以看到两个Pipe文件，它和普通文件不同，具有p属性，表明是一个管道文件。**



**2）Java中，使用文件读写的方式打开这两个文件，即可进行读写。**

##### **（3）信号（Signal）：**

信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送 信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。

##### **（4）消息（Message）队列：**

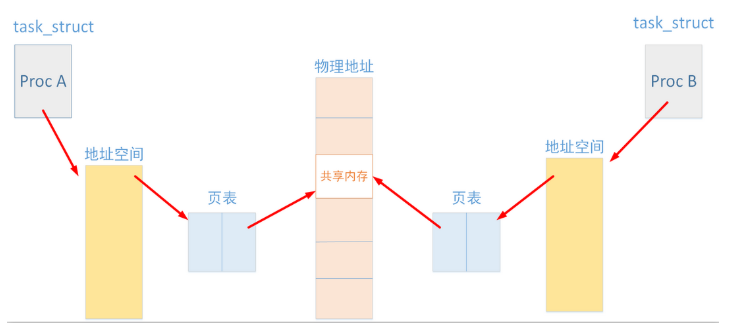
消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺

##### **（5）共享内存：**

使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

**在Linux中，每个进程都有属于自己的进程控制块（PCB）和地址空间（Addr Space），并且都有一个与之对应的页表，负责将进程的虚拟地址与物理地址进行映射，通过内存管理单元（MMU）进行管理。两个不同的虚拟地址通过页表映射到物理空间的同一区域，它们所指向的这块区域即共享内存。**

**共享内存的通信原理示意图：**



**对于上图我的理解是：当两个进程通过页表将虚拟地址映射到物理地址时，在物理地址中有一块共同的内存区，即共享内存，这块内存可以被两个进程同时看到。这样当一个进程进行写操作，另一个进程读操作就可以实现进程间通信。但是，我们要确保一个进程在写的时候不能被读，因此我们使用信号量来实现同步与互斥。**

**对于一个共享内存，实现采用的是引用计数的原理，当进程脱离共享存储区后，计数器减一，挂架成功时，计数器加一，只有当计数器变为零时，才能被删除。当进程终止时，它所附加的共享存储区都会自动脱离。**

##### **（6）内存映射（mapped memory）：**

内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。

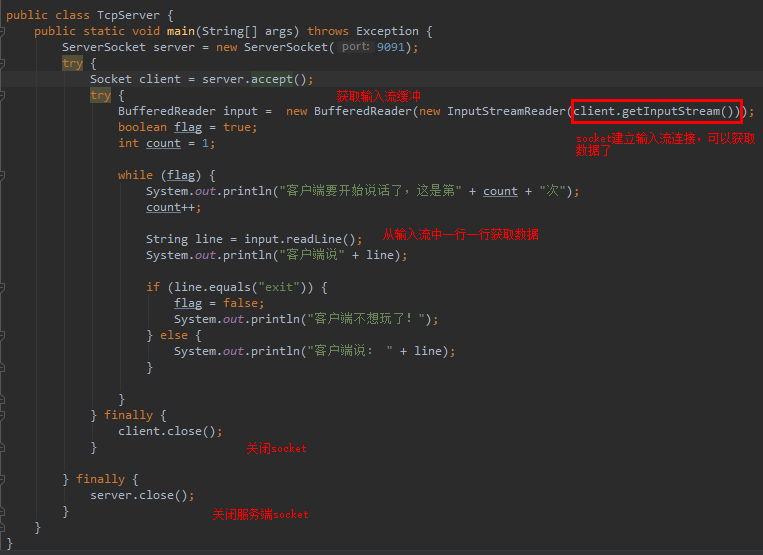
##### **（7）信号量（semaphore）：**

主要作为进程间以及同一进程不同线程之间的同步手段。

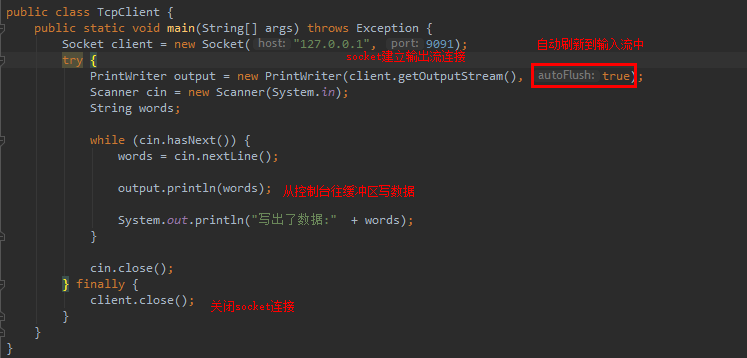
##### **（8）套接口（Socket）：**

更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

**服务端代码：**



**客户端代码**



## 多线程创建方法

一般来说是有五种方法创建线程：

1、直接实现Runnable接口，重写run方法，称为MyThread类

new Thread（new MyThread）.start()

2、继承Thread类，直接重写run方法

new Thread（），其中还是有一个target的Runnable对象

3、匿名内部类

new Thread( new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("匿名内部类的方式创建线程");  
 }  
 }).start();

4、实现Callable接口，重写其中的run方法，但是却要这样：

//FutureTask继承自RunnableFuture，而后者继承自Runnable

FutureTask<Integer> futureTask = new FutureTask<>(new C());  
 // 传递的仍是一个Runnable对象  
 Thread thread2 = new Thread(futureTask);  
 thread2.start();

5、采用线程池，就是存储上述方式创建的线程。

**综上：其实可以总结为一个方法，就是最终都要经过Thread这个类，传递给Thread类一个Runnable对象，然后调用Thread的start方法，有一个除外，无需传递Runnable对象，因为继承Thread类，自身携带target Runnable对象，其余的在构造函数时会调用init函数，该函数会将传入的Runnabe对象覆盖target对象。**

**关于你明明重写的是run方法，为什么调用的确实start方法。**

（1）start()方法是来自线程的，真正的实现了多线程的运行，而不用等到run()方法执行完毕之后再执行下面的代码段：

通过调用Thread类的**start()方法来启动一个线程，这时此线程是处于就绪状态**，并没有运行，**其实就是将该线程对象放入了线程组中group中。**  
 然后通过此Thread类调用方法run()来完成其运行操作的，这里方法run()称为线程体，它包含了要执行的这个线程的内容，**Run方法运行结束，此线程终止**，而CPU再运行其它线程。

（2）而run()方法只是类中我们自己定义的方法，是一种普通的方法，顺序执行，只有在执行完这一段代码之后才会继续执行下去，而**如果直接用Run方法，这只是调用一个方法而已，程序中依然只有主线程--这一个线程**，其程序执行路径还是只有一条，这样就没有达到写线程的目的。

## 用户态和内核态的理解和区别

首先明白的是这两种状态都是针对进程来说的，而进程是操作系统创建的，而每个进程所能做的工作都不一样，为了将这些工作分轻重缓急、重要程度，给进程建立了特权级的概念，最关键的工作交给特权级最高的线程去做，这样可以做到集中管理，减少有限资源的访问和冲突。而inter x86的cpu一共有4个级别0-3，其中0的特权级别最高。而用户态的进程就是特权级别为3，内核态的进程就是特权级别为0.

一般而言，自己刚开始创建的进程都会处于用户态，但如果要执行文件操作，网络数据发送等操作必须调用write\send等系统调用，那么这个时候需要将进程从用户态切换到内核态，因为用户态的进程的特权不允许访问本该内核态才能访问的代码和数据。当完成系统调用后，又会切换为用户态。

**总结一下：**

内核态：运行操作系统程序，操作硬件

**用户态：运行用户程序**

**差别在于：**

处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理器是可被抢占的

处于内核态执行时，则能访问所有的内存空间和对象，且所占有的处理器是不允许被抢占的。

# 26.输入输出流

## 字符流和字节流的对比

**继承关系如下：**

字节流：inputStream --> FileInputStream --> BufferInputStream、DataInputStream

outputStream --> FileOutputStream --> BufferOutputStream、DataOutputSteam

字符流：inputStreamReader--> FileReader --> BufferFileReader

InputSteamWriter --> FileWriter、Printwriter --> BufferFileWriter

FileInputStream在读取文件的时候，就是一个一个byte地读取，

DataInputStream则是在FileInputStream的一个轻量级的包装类，

BufferInputStream则是自带缓冲区，默认缓冲区大小为8X1024

通俗地讲就是：

FileInputStream在读取文件的时候，一滴一滴地把水从一个缸复制到另外一个缸

DataInputStream则是一瓢一瓢地把水从一个缸复制到另外一个缸

BufferInputStream则是一桶一桶地把水从一个缸复制到另外一个缸。

后两者，虽然底层封装的都是File流，但是明显buffer更厉害。

**一句话概括Buffer流的精髓：**

读数据的时候不需要频繁跟磁盘交互，先从buffer读，没有数据了，再从磁盘里拿；

写数据的时候不需要频繁和磁盘交互，先往buffer里面写，写满了，再刷到磁盘。

而且二者的buffer是共享的，这就导致了buffer里面都是热点数据，大大减少了IO操作。

注意，buffer里面存的都是原始的File输入输出流。

**FileInputStream与FileReader区别：**

FileInputStream是字节流，FileReader是字符流，用字节流读取中文的时候，可能会出现乱码，而用字符流则不会出现乱码，而且用字符流读取的速度比字节流要快；

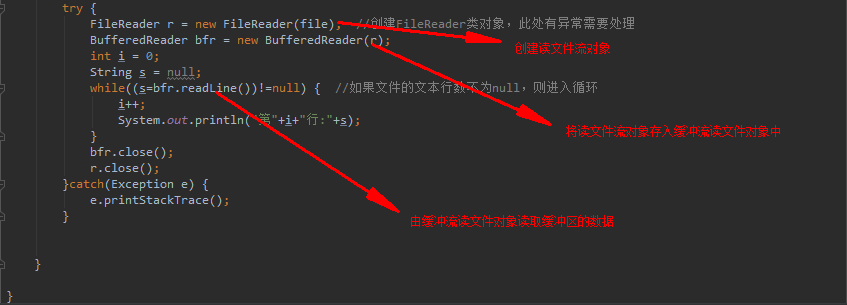
## 2.带缓冲的输入输出流（字符流流为例）

**BufferedReader和BufferedWriter类**

**写入文件过程：字符数据→BufferedWriter→FileWriter→文件**

**从文件中读出过程：文件→FileReader→BufferedReader→目的地**





# 27、锁的分类介绍（指的都是锁的思想）

**Java中就两种加锁的方式：**

**一种是使用synchronized关键字，另一种是使用Lock接口的实现类**

**前者就相当于自动挡，可以满足一切驾驶需求；但是你如果想要更高级的操作，比如玩漂移和各种高级的骚操作，那么就需要手动挡，也就是后者；**

**而Synchronized在经过每个版本的各种优化后，效率也变得很高了，只是使用起来没有Lock接口的实现类那么方便**

## 悲观锁和乐观锁

**乐观锁**：就是很乐观，每次去拿数据的时候都认为别人不会修改，如果想要更新数据，就会在更新之前检查在读取至更新这段时间别人有没有修改过这个数据，如果修改过，则重新读取，再次尝试更新，循环往复，直到更新成功，**当然也允许失败的线程放弃更新操作。这就是CAS（compare and swap）的概念**，是实现乐观锁的基础，**所以乐观锁是没有上锁的，所以乐观锁，也称为无锁编程。**

乐观锁允许多个线程同时读取（因为根本没有加锁操作），如果更新数据的话，有且仅有一个线程成功更新数据，并导致其他线程需要回滚重试，CAS利用CPU指令，从硬件层面保证了原子性，以达到类似于锁的效果。

**悲观锁：**就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次都在拿数据的时候上锁。想获取数据的线程，必须等到悲观锁释放，再去获得锁，然后再获取数据

**悲观锁阻塞事务，乐观锁回滚重试**

**适用场景**：乐观锁适用于写比较少的情况下，即冲突真的很少发生的场景，这样可以省去锁的开销，极大系统的整个吞吐量。如果经常发生冲突，应用就会不断重试，这样反而降低了性能，用悲观锁更好。

## 偏向锁、轻量级锁（自旋锁）、重量级锁（Sychronized为例）

前面讲了Synchronized锁已经经过了升级优化：就是不是直接上来就是悲观的重量级锁，而是从偏向锁 --> 轻量级锁 --> 重量级锁的转变。

也就是说使用sychronized关键字同步代码块的时候，上的是偏向锁，字面意思是**“偏向第一个获取它的线程的锁”**。**线程执行完同步代码块之后，并不会制动释放偏向锁**，当有线程第二次想进入同步代码块时，线程会判断持有锁的线程是否就是自己（持有锁的线程ID在锁的对象头里存储），如果是则正常往下执行。由于之前没有释放，这里不需要重新加锁，如果从头到尾都是一个线程在使用锁，很明显，偏向锁几乎没有额外的开销，这也说明适用synchronized关键字的锁是可冲入锁，

但是如果有第二个线程加入锁竞争，**偏向锁会转换为轻量级锁，也就是自旋锁**。**没有抢到锁的线程会进行自旋操作**，即在一个循环中不停判断是否可以获取锁，就是通过CAS操作修改对象头里的锁标志位，将锁的类型变为轻量级锁，如果抢到了锁，就把当前锁的所有者信息改为自己；

假如我们获取到锁的线程操作时间很长，那么其他线程的自旋时间较长，会发生忙等现象，如果忙等时间不长，可以用短时间的忙等获取线程在用户态和内核态之间切换的开销。

但是，**如果JVM中的PC计数器记录的自旋次数超过最大自旋次数，那么轻量级锁会升级为重量级锁**，依然是通过CAS修改锁标志位，表示达到了重量级锁级别，如果后续线程尝试获取锁，**发现被暂用的锁是重量级锁，那么直接将自己挂起，不会进行自旋操作**。

在JVM张，synchronized锁只能逐渐升级，而不允许降级。其中轻量级锁可以认为是加锁版的乐观锁。

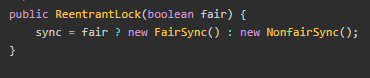
## 可重入锁（递归锁）

可以重新进入的锁，**即允许同一个线程多次获取同一把锁**，java中应该都是可重入锁，不可重入锁意义不大。

## 公平锁和非公平锁

如果多个线程申请一把公平锁，那么锁的想成释放锁时，**先申请的先得到，很公平，如果是非公平锁，后申请的线程可能先获得锁**，是随机还是其他方式，取决于实现算法。

对于ReentrantLock类来说，可以通过构造函数指定公平还是非公平。



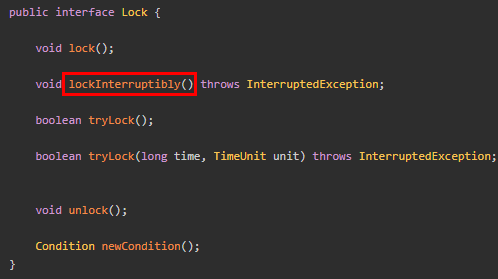
而对于synchronized锁而言，只能是非公平锁。

如果没有特殊要求，优先使用非公平锁，因为其吞吐量更大。

## 可中断锁

可以相应中断的锁，java中并没有提供任何可以直接中断线程的方法，只提供了中断机制，即Thread.interrupt()方法，但是不能直接终止线程，线程会自行选择在合适的时间点以自己的方式相应中断。

**Synchronized是不可中断锁， 而Lock的实现类都是可中断锁，**只要实现Lock接口的Interruptibly方法就好



## 读写锁

读写锁其实是一对锁，一个读锁（共享读锁）和一个写锁（互斥锁、排它锁）

如果我读取值是为了更新他（sql中的for update就是这个意思），那么加锁的时候直接加写锁，持有写锁时，别的线程，不论是读还是写，都需要等待；如果读取数据仅仅是为了前端展示，那么加锁时就明确加一个读锁，其他线程如果也要加读锁，不需要等待，可以直接获取（**读锁计数器加1**）。

虽然读写锁感觉像乐观锁，但是读写锁是悲观锁策略，**因为读写锁没有再更新前判断值有没有被修改过，而是在加锁前决定应用写锁还是读锁**。乐观锁特指无锁编程。

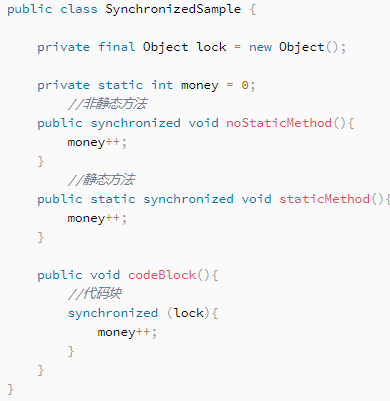
## synchronized详解

<https://blog.csdn.net/zhengwangzw/article/details/105141484>

**Synchronized有三种作用范围：**

1. 在静态方法上加锁
2. 在非静态方法上加锁
3. 在代码块上加锁

**注意：锁只能加在对象上。**



synchronized 在代码块上加锁是通过 monitorenter 和 monitorexit指令实现，在静态方法和非静态方法上加锁是在方法的flags 中加入 ACC\_SYNCHRONIZED 。JVM 运行方法时检查方法的flags，遇到同步标识开始启动前面的加锁流程，在方法内部遇到monitorenter指令开始加锁。

**JVM怎么通过sychronized在对象上加锁，保证多线程访问竟态资源安全。**

总的来说：

1. jdk6之前，都是重量级锁
2. Jdk6之后，引入了偏向锁和轻量级锁，
   1. 原因1、因为Sun 程序员发现大部分程序大多数时间都不会发生多个线程同时访问竞态资源的情况，每次线程都加锁解锁，每次这么搞都要操作系统在用户态和内核态之间来回切（加锁后，线程处于内核态，不允许抢占该线程占有的资源），太耗性能了。线程获取资源的时候，会发生系统调用，从用户态转化为内核态，假如加了重量锁，线程会重返用户态，来回切换会消耗性能，而假设加了轻量锁，线程会先保持内核态，当自旋操作达到一定数量后，将轻量锁标志位置为重量锁标志位，然后再重返用户态。
   2. 原因2、同一个线程加锁解锁的重复率很高，这也是偏向锁的由来。

偏向锁适合低并发时，同一个线程总是获得该资源，为避免重复的加锁解锁操作，而保持的一直占用，当高并发时，用的都是轻量级锁，因为线程执行的时间可能很短亦或是很少发生线程争抢，所以加轻量级锁就好，使进程保持短暂的内核态比用户态和内核态切换消耗的性能要少，当线程执行时间比较长或者并发度很高时，那么使用重量级锁较好。

**总结：**

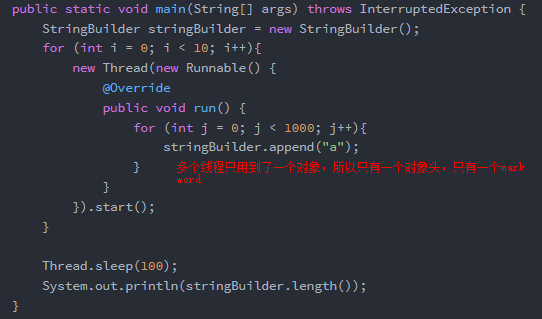
并发度极低：使用偏向锁，避免频繁加锁解锁

并发度中等，线程执行时间较短：使用轻量级锁，使进程保持短暂的内核态，避免频繁的用户态和内核态切换。

并发度高或线程执行时间较长：使用重量级锁，防止大量线程执行较长时间的轻量级锁的自旋操作。

加锁具象化理解：

在一个类中，全局变量可以看做是竟态资源，如StringBuffer当中的value数组， 而对其中的append方法加sychronized关键字，实例化一个StringBuffer对象，在多个线程中调用StringBuffer对象的append方法，锁住的就是当前对象，不允许多个线程同时调用append方法，那么也就不会同时操作value数组。也就是说，**sychronized锁住的其实是操作资源的代码块，而不是真的资源。**



## 8.java对象头

Java 对象头又是什么呢？我们以 Hotspot 虚拟机为例，Hopspot 对象头主要包括两部分数据：Mark Word（标记字段） 和 class Pointer（类型指针）。

Mark Word：默认存储对象的HashCode，分代年龄和锁标志位信息。这些信息都是与对象自身定义无关的数据，所以Mark Word被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据。它会根据对象的状态复用自己的存储空间，也就是说在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。

class Point：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。



无状态也就是无锁的时候，对象头开辟 25bit 的空间用来存储对象的 hashcode ，4bit 用于存放分代年龄，1bit 用来存放是否偏向锁的标识位，2bit 用来存放锁标识位为01

偏向锁 中划分更细，还是开辟25bit 的空间，其中23bit 用来存放线程ID，2bit 用来存放 epoch，4bit 存放分代年龄，1bit 存放是否偏向锁标识， 0表示无锁，1表示偏向锁，锁的标识位还是01

轻量级锁中直接开辟 30bit 的空间存放指向栈中锁记录的指针，2bit 存放锁的标志位，其标志位为00

重量级锁中和轻量级锁一样，30bit 的空间用来存放指向重量级锁的指针，2bit 存放锁的标识位，为11

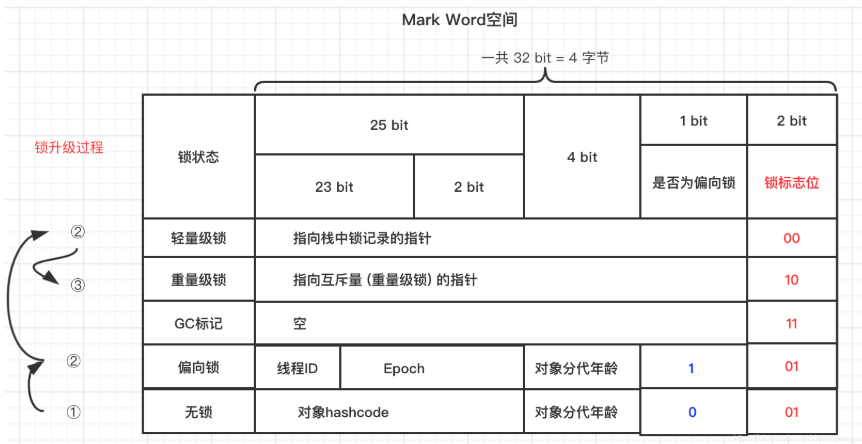
GC标记开辟30bit 的内存空间却没有占用，2bit 空间存放锁标志位为11。

其中无锁和偏向锁的锁标志位都是01，只是在前面的1bit区分了这是无锁状态还是偏向锁状态。

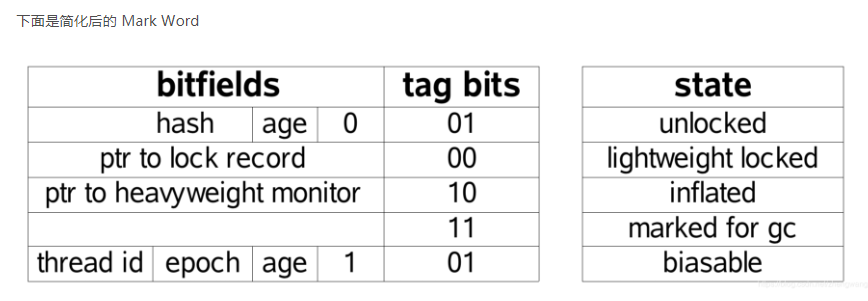
首先，java对象包含三个部分：

**java对象头：**





以上是32位JVM背景下的对象头中的mark word，锁标志位代表该对象是否获得该资源，获的该资源加的锁是什么，01是初始化时的标志位，表示无锁，年龄为1时，表示获得该资源，加了偏向锁，00是轻量级锁，10是重量级锁，11表示该对象要被回收。



**对象实例数据**：

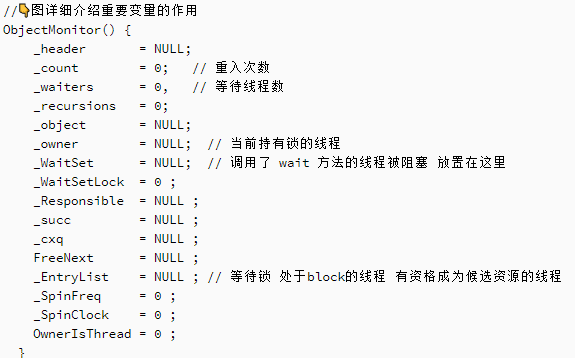
成员变量等

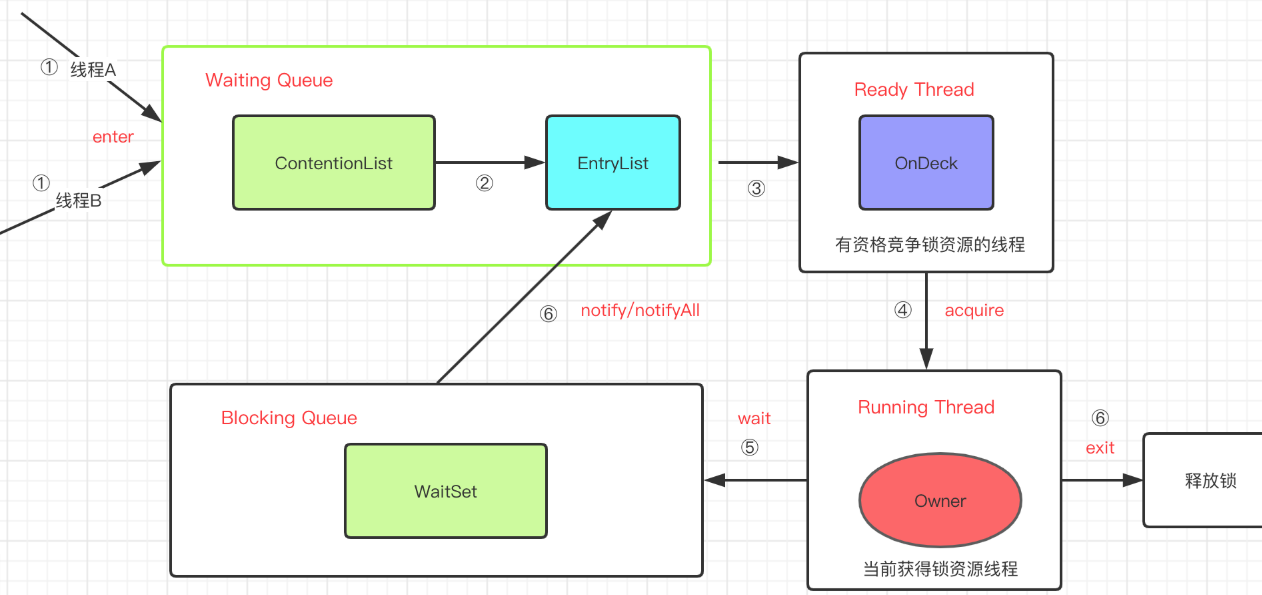
**对齐填充：**

JVM要求对象占用的空间必须是8 的倍数，方便内存分配（以字节为最小单位分配），因此这部分就是用于填满不够的空间凑数用的。

## 9.Monitor

每个对象都有一个与之关联的Monitor 对象

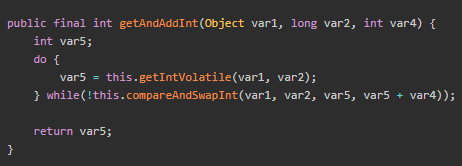




## 总结

Java中的各种锁基本都是悲观锁，乐观锁的话只有java.uti.concurrent.atomic下面的原子类是通过乐观锁实现的。

例如：



可以看到，在一个循环里不断CAS，知道成功为止。