Sed 命令，利用脚本来处理文本文件
语法，

```
sed [-hnV][-e<script>][-f<script 文件>][文本文件]
```

参数说明：

- -e<script>或--expression=<script> 以选项中指定的 script 来处理输入的文本文件。
- -f<script 文件>或--file=<script 文件> 以选项中指定的 script 文件来处理输入的文本文件。
- -h 或--help 显示帮助。
- -n 或--quiet 或--silent 仅显示 script 处理后的结果。
- -V 或--version 显示版本信息。

打印第十行例子：
Sed -n 10p file.txt

Grep 命令，查找文件里符合条件的字符串
匹配电话号码例子：
grep -e '^[0-9]\{3\}-[0-9]\{3\}-[0-9]\{4\}$' -e '^([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}$' file.txt

什么是 override?
当子类从父类继承一个无参方法，而又定义了一个同样的无参数的方法时，则子类新写的方法覆盖父类的方法，成为覆盖

Overload?
继承了父类的同名有参函数:当子类继承了父类的一个同名参数，且方法参数表不同，则称为重载，通过方法重载，子类可以重新实现父类的某些方法，使其具有自己的特征

```
class Super{
    public void WithParametersMethod(int a){
        System.out.println("Super Method:"+a);
    }
}

class subClass extends Super {
    public void WithParametersMethod(int a,int b){
```

```
        System.out.println("subClacc Method:"+a+","+b);
    }
}
```

Overwrite?
类中同名方法，不同参数。

多态就是同一个接口，使用不同的实例而执行不同操作

进程和线程
1) Both process and Thread are independent path of execution but one process can have multiple Threads.
一个 process 可以拥有很多 thread，反之不行;

2) Every process has its own memory space, executable code and a unique process identifier (PID) while every thread has its own stack in Java but it uses process main memory and share it with other threads.
process 会有自己的 memory space，code, PID, thread 会有自己的 stack，但是 thread, share process 的 file descriptors，heap memory。去和 process 中其他的 thread 进行通信;

3) Threads are also refereed as task or light weight process (LWP) in operating system
thread 是 light weight process

4) Threads from same process can communicate with each other by using Programming language construct like wait and notify in Java and much simpler than inter process communication.
同一个 process 中的 thread，可以通过 wait notify programming language 去通信;

5) Another difference between Process and Thread in Java is that it's How Thread and process are created. It's easy to create Thread as compared to Process which requires duplication of parent process.
thread 很容易创建，而 process 需要复制 parent process;

并发（concurrency）和并行（parallellism）
并发 一个人吃三个馒头
并行，三个人同时吃三个馒头

- 解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。
- 解释二：并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。
- 解释三：并行是在一台处理器上“同时”处理多个任务，并发是在多台处理器上同时处理多个任务。如 hadoop 分布式集群。

并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。所以无论从微观还是从宏观来看，二者都是一起执行的。

并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行

内存泄露：

Java 中的内存泄露，广义并通俗的说，就是：不再会被使用的对象的内存不能被回收，就是内存泄露

```
public class Simple {

    Object object;

    public void method1(){
        object = new Object();
    //...其他代码
    }
}
```

这里的 object 实例，其实我们期望它只作用于 method1()方法中，且其他地方不会再用到它，但是，当 method1()方法执行完成后，object 对象所分配的内存不会马上被认为是可以被释放的对象，只有在 Simple 类创建的对象被释放后才会被释放，严格的说，这就是一种内存泄露。解决方法就是将 object 作为 method1()方法中的局部变量。当然，如果一定要这么写，可以改为这样：

```
public class Simple {

    Object object;

    public void method1(){
        object = new Object();
```

```
        //...其他代码
        object = null;
    }
    }
```

MySQL 的 left join , right join and inner join

1 a20050111

2 a20050112

3 a20050113

4 a20050114

5 a20050115

表 B 记录如下:

bID bName

1 2006032401

2 2006032402

3 2006032403

4 2006032404

8 2006032408

Left join 结果

aID aNum bID bName

1 a20050111 1 2006032401

2 a20050112 2 2006032402

3 a20050113 3 2006032403

4 a20050114 4 2006032404

5 a20050115 NULL NULL

right join 结果

aID aNum bID bName

1 a20050111 1 2006032401

2 a20050112 2 2006032402

3 a20050113 3 2006032403

4 a20050114 4 2006032404

NULL NULL 8 2006032408

inner join

aID aNum bID bName

1 a20050111 1 2006032401

2 a20050112 2 2006032402

3 a20050113 3 2006032403

4 a20050114 4 2006032404

128 十进制-》7 进制? 只能有 0-6

128/100=1 第一位是 1

128-100=28

28/10=2 第二位是 2

28-2*10=8

8/1=8

第三位是 8

128/49=2 余数 30

第一位是 2

128-2*49=30

30/7=4 余数 2

第二位是 4

30-28=2

第三位是 2

242 2*49+4*7+2=98+28+2

Awk 命令，

```
name age

alice 21

ryan 30
```

例如转置此文件

Awk 默认按照行执行，NR 是行数，NF 是列数， &i 就是取出内容。

```
awk '
{
    for (i = 1; i <= NF; i++){
        if (NR == 1){
            s[i] = $i;
        }else{
            s[i] = s[i] " " $i;
        }
    }
}
END{
    for (i = 1; s[i] != ""; i++){
        print s[i];
    }
```

}' file.txt

数据库的 normalization

第一范式 1NF（1st normal form）

不允许表中套表

name	tel		age
大宝	13612345678		22
小明	13988776655	010 - 1234567	21

纠正方法：应该是 name,tel,age　和 tel,cell phone, fix_phone 两张表

第二范式 2NF

学生	课程	老师	老师职称	教材
小明	一年级语文（上）	大宝	副教授	《小学语文 1》

符合 1NF 的基础上，非主属性完全函数依赖于主属性，适用于联合主键的表。

例如学生，课程是联合主键，但是　!!!　只要有课程就已经有对应的教材了，所以不完全依赖。

解决方法：学生，课程，老师，职称，教室，上课时间。

另外一张表是：　课程，教材。

3NF

第三范式

在符合 2NF 基础上，消除传递函数依赖，transitive dependency

存在（学生，课程）-》老师-》职称　传递依赖关系：

修改方法，将老师也分离出来，建立老师，职称表。

BC 范式：主属性不依赖于主属性。

数据库索引是对数据库表中一列或者多列进行排序的结构，可以加快搜索速度，但是增加了存储空间，另外插入修改数据时也会花费更多时间。

HTTP 中的 GET 和 POST 有什么区别?

HTTP 定义了与服务器交互的四种方法：get post, put, delete 描述了 CRUD 操作。GET 一般用来获取查询资源信息，POST 用于更新资源信息。

Get 只是获取，不会修改增加数据，不会影响资源的状态，并且多次请求返回相同结果。

Post 用于向服务器提交数据，比如表单数据提交 login，将数据交给服务器处理。

Get 请求的数据会放在 URL 后面，post 把提交的数据放在 HTTP 包的包体中，post 安全性更高，防止了 CSRF 攻击（cross -site-request-forgery 攻击）

Java 的 access modifier

Public：被所有类访问

Protected：类内部，相同包，以及该类的子类可以访问。

Private：只能在类内部访问

Default：类内部以及相同包内使用。

抽象类和接口的比较

参数	抽象类	接口
默认的方法实现	可以有默认的方法实现	完全抽象，根本不存在方法的实现
实现方式	子类用 **extends** 关键字来继承抽象类，如果子类 不是抽象类的话，它需要实现父级抽象类中所有抽 象方法，父类中非抽象	子类用 **implements** 去实现接口，需要实现接口中所有方法

参数	抽象类	接口
	方法可重写也可不重写	
构造器	抽象类可以有构造器（构造器不能用 abstract 修饰）	接口不能有构造器
与正常 Java 类的区别	正常 Java 类可被实例化，抽象类不能被实例化，其他区别见上下文	接口和正常 java 类是不同的类型
访问修饰符	抽象方法可以用 public、protected、default 修饰	接口默认是 public、不能用别的修饰符去修饰
main 方法	抽象类中可以有 main 方法，可以运行它	接口中不能有 main 方法，因此不能运行它
多继承	抽象类可继承一个类和实现多个接口	接口只能继承一个或者多个接口
速度	抽象类比接口速度快	接口稍微慢点，因为它需要去寻找类中实现的它的方法
添加新方法	如果在抽象类中添加新非 abstract	只要在接口中添加方法，实现它的类就要改变，去实现这个新添加的方法

参数	抽象类	接口
	的方法，可以直接添加，因为非 abstract 方法无需在子类中实现，如果是 abstact 方法，则需要改变子类的代码，也要实现这个方法	

接口和抽象类分别在什么时候使用

a. 如果拥有一些方法，并想让他们中的一些有默认的具体实现，请选择抽象类

b. 如果想实现多重继承，那么请使用接口，由于 java 不支持多继承，子类不能继承多个类，但一个类可以实现多个接口，因此可以使用接口来解决。

c. 如果基本功能在不断变化，那么就使用抽象类，如果使用接口，那么每次变更都需要相应的去改变实现该接口的所有类。

Hashmap 13 14 节课

Cache 就是一个临时存储空间，容量小，交换速度快，如何 cache，两种 space locality 和 temporary locality

经常被读取，读取频率远远大于改写频率的数据可以缓存

变化非常少的数据 或者使用率非常高的数据 可以缓存。

Web 缓存通过保留 HTTP 响应和 web 资源。

1.client side web caching

2.server-side：Reverse proxy caches or web application accelerators 放在浏览器和 origin server 之间。它缓存的是 HTTP 请求对应的 HTTP 响应

另一种使用键值对存储，例如 memcached,和 redis，它可以存储网页内容，存储 web sessions。比如亚马逊 ElasticCache 就是用了键值对存储。

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\\\\"匹配"\\"，"\\("匹配"("。
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配"z"和"zoo"。* 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，但与"z"不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{*n*,*m*}	*m* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（*、+、?、{*n*}、{*n*,}、{*n*,*m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
.	匹配除"\r\n"之外的任何单个字符。若要匹配包括"\r\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。

(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"\("或者"\)"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。
(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的 "Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	匹配 *x* 或 *y*。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。
[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。

\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 [^0-9]。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
n	标识一个八进制转义码或反向引用。如果 *n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。

\nm	标识一个八进制转义码或反向引用。如果 \nm 前面至少有 nm 个捕获子表达式，那么 nm 是反向引用。如果 \nm 前面至少有 n 个捕获，则 n 是反向引用，后面跟有字符 m。如果两种前面的情况都不存在，则 \nm 匹配八进制值 nm，其中 n 和 m 是八进制数字 (0-7)。
\nml	当 n 是八进制数 (0-3)，m 和 l 是八进制数 (0-7) 时，匹配八进制转义码 nml。
\un	匹配 n，其中 n 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)。

Throw 和 Throws

Throw 用来抛出异常，抛出的异常必须是 Throwable 类型或者它的子类。

Throws?

在函数 signature 中使用，指出可能抛出的异常。

```
static void fun() throws IllegalAccessException{
        System.out.println("inside fun().");
        throw new IllegalAccessException("demo");

    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try {
            fun();
        }catch(IllegalAccessException e) {
            System.out.println("Caugut in main");
        }
        }
```

泛型类

```
public class Generic<T>{
    private T key;

    public Generic(T key) {
        this.key = key;
    }

    public T getKey(){
        return key;
    }
    }
```

Garbage collection 机制是什么？

Manage dynamic memory allocation more efficiently

一个 JVM 只有一个堆，一个 thread 就有一个栈。

堆报错，outOfMEemoryError

栈报错　Stack Overflowerror

栈内存远远小于堆内存

Automatic GC(使用 garbage collector)

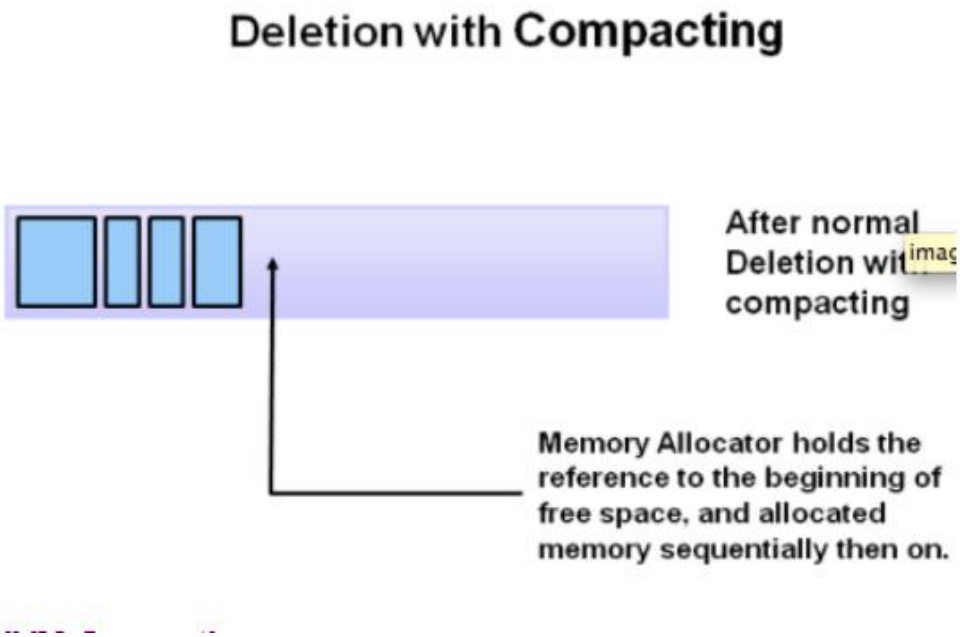
第一步 marking

将 unused 的 object 标记

第二步 normal deletion

将 unused 的 object 移除，memory allocator 记录可用空间 reference

并且将剩余 referenced object compact



JVM 的 generation

Young , old, permanent generation

每次 runGC 时候，就要停止 main java program，因此 delay of response 是一个问题，这叫做 stop the world event.

为什么使用 JVM generations?

堆很大，不想 stop world 时间太长。

Generation 根据 object 的生命周期长度定义，GC selector 只对长生命周期的 object 运行 GC

新生代通过 minor GC 到了 old generation,=》major GC 到了 permanent generation

永久代包含 metadata 用来描述类和方法，还有 Java SE library class and methods

如何知道是否这个 object 需要 GC（DFS）

定义 dependency

Local variable 和 active java threads，static variable，java native interface 都是 GC Root。

Test 有哪些？？？

Unit test，integration test，regression test, smoke test, end-2-end test, black/white test, performance test…

对于 JUNI test 里，可用的 utils 有

@Test @Before,@After,@BeforeClass,@AfterClass

Assertequals,asserttrue,assertnotnull,assertnull,assertsame,assertarrayequals…

不同线程，每个线程都有各自的栈，program counter，

但是共用堆，static memory segment，os resource。

Thread t = new Thread();

t.start();

t.join();

daemon thread=>用户线程

yield 停止该线程，并允许其它线程执行。

什么是 data race： >=1 operations work on the same memory location，至少一个操作是写，并且至少两个操作是并发的。

如何解决 data race?

加锁，语义上使用 mutual exclusion

例子， 停车场停车，订票选座。。。。。。。

通常支持两个操作，lock 和 unlock

Lock，等待没别人了，进入，

Unlock，离开，并且标记这里没人了

实现上，使用 synchronized 关键词

对非静态方法的同步==同步 this。

对静态的同步！=同步所有 instance

如果使用普通的 boolean 锁，两个 thread 会同时看到 boolean value，并且会同时不运行！！！。

实际上应该 A 锁了运行 B， B 锁了运行 A， 结果 AB 全假设锁了，都不运行了。。

死锁是什么?

Lock1 锁住自己，试图进入 Mutex 的 Lock2！！！两个以上进程执行过程中，由于竞争资源或者彼此通信造成阻塞的现象。之后永远处于相互等待的状态。

产生死锁的四个条件：

1. Mutual exclusion
2. No preemption:进程获得的资源在未使用完毕之前，不能被其它进程夺走，只能自己释放。
3. Hold and wait:进程已经获得了资源，但是还需要其他资源，这些资源被其它进程控制，但进程对其控制不放。

4. 循环等待 circular wait：链中每一个进程获得的资源同时被下一个进程请求。

活锁：俩人对着走，一人向左一人向右 都为了让对面通过，happen when each side actively resolving the problem.

活锁有可能自行解开。

解决方法，加入优先级和随机性!!!

Java 中的可见性，Atomicity 原子性和有序性。

visibility 可见性是说一个线程修改的状态对另一个线程是可见的，例如用 volatile 修饰的变量具有可见性。但是不能保证原子性。具有可见性是因为修饰的变量不允许线程内部缓存和重新排序。

Volatile synchronized 和 final 都具有可见性!!!

A=0 是原子操作，但是 A++不是原子操作，需要加入 sychronized

使用 sychronized 和 lock unlock 实现原子性。

有序性，volatile 不允许线程内部重新排序，而且 sychronized 不运行多个线程对同一个变量进行 lock 操作，因为持有同一个对象锁的两个同步块只能串行执行。

\

Thread.wait()

等待其它线程的某些资源，才继续执行。

=》 为了 efficiently，需要 conditional synchronization

Ideally， when a thread satisfies the condition to notify all of the waiting threads

三个操作：wait()　notify()　notifyAll()

生产消费者模型：

建立一个 blocking queue，消费者等到 queue 不空，然后拿一个。生产者等到 queue 不满，

然后放上去一个！！！

Mutual exclusive：使用 lock

Wait：　=》 conditional synchronization

什么是 monitor？实现方法是两个 queue

一个 lock queue 每次只能进一个 thread，一个是 wait queue，每次可以把所有 wait 的

thread 都装进去，然后等待 notifyall();

Readwrite lock：为什么使用读写锁?

Synchronized 的一个最大问题就是读与读之间排斥，而我们需要只有写的时候才排斥。

读写锁是一个接口，里面有两个方法：readlock 和 writelock

ReetrantReadWriteLock 实现了读写锁接口!!!

非公平模式获取锁顺序：竞争获取，会延缓一个或多个读或写线程，但是有更高的吞吐量。
公平模式，以队列的顺序。等待时间最长的写锁线程会被分配写锁。

重入锁：一个线程获取某个锁后，还可以继续获取。例子：synchronized 内置锁可重入。A 类两个 synch 方法，那么 method1 可以调用 method2.

Reentrantlock 也是可重入锁。

不支持锁升级，也就是读锁变成写锁，属于死锁。

但是支持锁降级，也就是写锁变成读锁，但是仍然需要显示释放写锁。

```
public class ReadAndWriteLockTest {
    // because two thread both read operation, so not exclusive if use readwrite
lock!!
    // for RW lock, read lock is share mode, and write lock is self mode
    // read and write lock is exclusive
    // write and write is exclusive
    public static void get(Thread thread) {
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        lock.readLock().lock();
        System.out.println("start time:" + System.currentTimeMillis());
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(20);
            }catch(InterruptedException e) {
                e.printStackTrace();
```

```
            }
            System.out.println(thread.getName() + ": is reading...");
        }
        System.out.println(thread.getName() + "read completed");
        System.out.println("end time:" + System.currentTimeMillis());
        lock.readLock().unlock();
    }

    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                get(Thread.currentThread());
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                get(Thread.currentThread());
            }
        }).start();
    }

}
```