1. 序列化的id有什么作用

每个实体化对象类如果没有声明序列化id，虚拟机就会根据这个对象的属性，计算出一个序列化id出来，如果更改对这个对象删除或添加属性，这个序列化的id也会改变，所以在网络中传输一个对象的时候，必须写死这个对象的序列化id，否则有一端序列化对象的属性改变了，则另外一端反序列化的时候就会报错

1. hashMap的存储结构

HashMap 是用数组（存放key-value）+链表（存放value）的形式进行存储，hashMap类型声明了一个长度为16的table数组，这个table数组中存放的是key-value形式的entry对象

插入过程：

比如map.put(“a”,”aaa”);

其会根据a的hash值计算出a-aaa的entry对象要存放在table数组的下标位置

取出过程

比如map.get(“a”)

其会根据a的hash值，根据hash值反计算出，a-aaa所存在table数组的下标位置，从而取出entry对象，获取到aaa这个值

1. list，set存储结构

List:

线性表的顺序存储，存放的顺序是什么，取出来的顺序也是什么，所以允许存入的

值重复

Set:

线性表的hash存储，存放引用的地址是根据值的hash值算出来的，所以相同元素的hash值相同，所以存入的元素的是唯一的，取出来的时候是根据hash值的从小到大排序查出来的

4、心跳机制的代码实现

服务器端：起一个线程接收并返回

客户端：一个线程每隔2s发送数据，实时的接收服务端返回的数据

是通过socket的tcp通信建立起来的

5、怎样提高io的读或写的速度

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 流分类 | 使用分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
|  | 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 节点流 | 访问文件 | **FileInputStream** | **FileOutStream** | **FileReader** | **FileWriter** |
| 访问数值 | **ByteArrayInputStream** | **ByteArrayOutStream** | **CharArrayReader** | **CharArrayWriter** |
| 访问管道 | **PipedInputStream** | **PipedOutStream** | **PipedReader** | **PipedWriter** |
| 访问字符串 |  |  | **StringReader** | **StringWriter** |
| 处理流 | 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 |  |  | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream |  |  |
| 抽象基类（过滤） | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 |  | PrintStream |  | PrintWriter |
| 推回输入流 | PushbackInputStream |  | PushbackReader |  |
| 特殊流 | DataInputStream | DataOutputStream |  |  |

总结：

1.Java IO是采用的是装饰模式，即采用**处理流**来包装**节点流**的方式，来达到代码通用性。

2.处理流和节点流的区分方法，**节点流**在新建时需要一个数据源（文件、网络）作为参数，而**处理流**需要一个节点流作为参数。

3.**处理流**的作用就是提高代码通用性，编写代码的便捷性，提高性能。

4.**节点流**都是对应抽象基类的实现类，它们都实现了抽象基类的基础读写方法。其中read（）方法如果返回-1，代表已经读到数据源末尾。

|  |
| --- |
| **package** test;  **import** java.io.BufferedInputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.nio.ByteBuffer;  **import** java.nio.channels.FileChannel;  **public** **class** OldIoSpeed {  **private** **static** String *filePath* = "D:/myiotestfile.txt"; // 24m大小  **public** **static** **void** readOneByOne() **throws** IOException{  **long** time1= System.*currentTimeMillis*();  FileInputStream in = **new** FileInputStream(**new** File(*filePath*));  **int** b=0;  **while** ((b = in.read()) != -1) { //\*\* FileInputStream\*\*\*\*的read方法每次\*\*读取文件一个字节    }  in.close();  **long** time2= System.*currentTimeMillis*();  System.*out*.println(time2 - time1);  }  **public** **static** **void** readOneByAll() **throws** IOException{  **long** time1= System.*currentTimeMillis*();  File file = **new** File(*filePath*);  FileInputStream in = **new** FileInputStream(file);  **int** b=0;  **byte** buf[] = **new** **byte**[(**int**)file.length()];  **while** ((b = in.read()) != -1) { //\*\* FileInputStream\*\*\*\*的read方法每次\*\*读取文件一个字节    }  **long** time2= System.*currentTimeMillis*();  System.*out*.println(time2 - time1);  }  **public** **static** **void** readByBuffered() **throws** IOException{  **long** time1= System.*currentTimeMillis*();  FileInputStream in = **new** FileInputStream(**new** File(*filePath*));  BufferedInputStream bs = **new** BufferedInputStream (in);  **int** b=0;  **while** ((b = bs.read()) != -1) { //\*\* FileInputStream\*\*\*\*的read方法每次\*\*读取文件一个字节    }  **long** time2= System.*currentTimeMillis*();  System.*out*.println(time2 - time1);  }  **public** **static** **void** readByArray() **throws** IOException{  **long** time1= System.*currentTimeMillis*();  FileInputStream in = **new** FileInputStream(**new** File(*filePath*));  **byte** buf[] = **new** **byte**[2048];  **int** b=0;  **while** ((b = in.read(buf)) != -1) { //\*\* FileInputStream\*\*\*\*的read方法每次\*\*读取文件一个字节    }    **long** time2= System.*currentTimeMillis*();  System.*out*.println(time2 - time1);  }  **public** **static** **void** readByArrayAndBuffered() **throws** IOException{  **long** time1= System.*currentTimeMillis*();  FileInputStream in = **new** FileInputStream(**new** File(*filePath*));  BufferedInputStream bs = **new** BufferedInputStream (in);  **int** b=0;  **byte** buf[] = **new** **byte**[2048];  **while** ((b = bs.read(buf)) != -1) { //\*\* FileInputStream\*\*\*\*的read方法每次\*\*读取文件一个字节    }  **long** time2= System.*currentTimeMillis*();  System.*out*.println(time2 - time1);  }  **public** **static** **void** readByNio() **throws** IOException{  **long** time1= System.*currentTimeMillis*();  FileInputStream in = **new** FileInputStream(**new** File(*filePath*));  FileChannel channel = in.getChannel();  //分配 16KB缓冲区  ByteBuffer bb = ByteBuffer.*allocate*(1024<<4);    **while**(channel.read(bb)>0){  // 缓冲区翻转用于输出到foc  // bb.flip();  // 清空缓冲区用于下次读取  bb.clear();  }  **if**(**null**!=in)  in.close();  **if**(**null**!=channel)  channel.close();    **long** time2= System.*currentTimeMillis*();  System.*out*.println(time2 - time1);  }      **public** **static** **void** main(String[] args) **throws** IOException {  // readOneByOne(); // 91629  // readOneByAll(); //90213  // readByBuffered(); //622ms  // readByArray(); //93ms  // readByArrayAndBuffered(); //31ms  // readByNio();//30  //加了缓冲区后，合理调节缓冲区的大小，会很有效的提高效率  }  } |

1. concurrentHashMap跟newio用到差不多原理提高写入和读出效率

concurrentHashMap 会初始化16段，每段会有锁，所以支持16个线程并发操作，

存入的时候是根据key的hash来确定是哪一段，取出的时候，是根据key的hash值来去确定从哪一段取出

hashTable 整个对象只有一把锁，所以只能支持一个线程使用

1. super,volalite,synchorined,this,thranct关键字

Volalite :内存可见性：通俗来说就是，线程A对一个volatile变量的修改，对于其它线程来说是可见的，即线程每次获取volatile变量的值都是最新的

通过关键字sychronize可以防止多个线程进入同一段代码，在某些特定场景中，volatile相当于一个轻量级的sychronize，因为不会引起线程的上下文切换，但是使用volatile必须满足两个条件：

　　1、对变量的写操作不依赖当前值，如多线程下执行a++，是无法通过volatile保证结果准确性的;

　　2、该变量没有包含在具有其它变量的不变式中，这句话有点拗口，看代码比较直观。

Synchorined:

synchronized是Java中的关键字，是一种同步锁。它修饰的对象有以下几种：   
1. 修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；   
2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；   
3. 修改一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；   
4. 修改一个类，其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

# **修饰一个代码块**

1、一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞。我们看下面一个例子：

****【Demo1】：synchronized的用法****

/\*\*

\* 同步线程

\*/

class SyncThread implements Runnable {

private static int count;

public SyncThread() {

count = 0;

}

public void run() {

synchronized(this) {

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

try {

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

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public int getCount() {

return count;

}

}

SyncThread的调用：

SyncThread syncThread = new SyncThread();Thread thread1 = new Thread(syncThread, "SyncThread1");Thread thread2 = new Thread(syncThread, "SyncThread2");

thread1.start();

thread2.start();

结果如下：

SyncThread1:0   
SyncThread1:1   
SyncThread1:2   
SyncThread1:3   
SyncThread1:4   
SyncThread2:5   
SyncThread2:6   
SyncThread2:7   
SyncThread2:8   
SyncThread2:9\*

当两个并发线程(thread1和thread2)访问同一个对象(syncThread)中的synchronized代码块时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞，必须等待当前线程执行完这个代码块以后才能执行该代码块。Thread1和thread2是互斥的，因为在执行synchronized代码块时会锁定当前的对象，只有执行完该代码块才能释放该对象锁，下一个线程才能执行并锁定该对象。   
我们再把SyncThread的调用稍微改一下：

Thread thread1 = new Thread(new SyncThread(), "SyncThread1");Thread thread2 = new Thread(new SyncThread(), "SyncThread2");

thread1.start();

thread2.start();

结果如下：

SyncThread1:0   
SyncThread2:1   
SyncThread1:2   
SyncThread2:3   
SyncThread1:4   
SyncThread2:5   
SyncThread2:6   
SyncThread1:7   
SyncThread1:8   
SyncThread2:9

不是说一个线程执行synchronized代码块时其它的线程受阻塞吗？为什么上面的例子中thread1和thread2同时在执行。这是因为synchronized只锁定对象，每个对象只有一个锁（lock）与之相关联，而上面的代码等同于下面这段代码：

SyncThread syncThread1 = new SyncThread();

SyncThread syncThread2 = new SyncThread();

Thread thread1 = new Thread(syncThread1, "SyncThread1");

Thread thread2 = new Thread(syncThread2, "SyncThread2");

thread1.start();

thread2.start();

这时创建了两个SyncThread的对象syncThread1和syncThread2，线程thread1执行的是syncThread1对象中的synchronized代码(run)，而线程thread2执行的是syncThread2对象中的synchronized代码(run)；我们知道synchronized锁定的是对象，这时会有两把锁分别锁定syncThread1对象和syncThread2对象，而这两把锁是互不干扰的，不形成互斥，所以两个线程可以同时执行。

2.当一个线程访问对象的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该对象中的非synchronized(this)同步代码块。   
****【Demo2】：多个线程访问synchronized和非synchronized代码块****

class Counter implements Runnable{

private int count;

public Counter() {

count = 0;

}

public void countAdd() {

synchronized(this) {

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

try {

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

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

//非synchronized代码块，未对count进行读写操作，所以可以不用synchronized

public void printCount() {

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

try {

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

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public void run() {

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

if (threadName.equals("A")) {

countAdd();

} else if (threadName.equals("B")) {

printCount();

}

}

}

调用代码:

Counter counter = new Counter();Thread thread1 = new Thread(counter, "A");Thread thread2 = new Thread(counter, "B");

thread1.start();

thread2.start();

结果如下：

A:0   
B count:1   
A:1   
B count:2   
A:2   
B count:3   
A:3   
B count:4   
A:4   
B count:5

上面代码中countAdd是一个synchronized的，printCount是非synchronized的。从上面的结果中可以看出一个线程访问一个对象的synchronized代码块时，别的线程可以访问该对象的非synchronized代码块而不受阻塞。

1. 指定要给某个对象加锁

****【Demo3】:指定要给某个对象加锁****

/\*\*

\* 银行账户类

\*/

class Account {

String name;

float amount;

public Account(String name, float amount) {

this.name = name;

this.amount = amount;

}

//存钱

public void deposit(float amt) {

amount += amt;

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

//取钱

public void withdraw(float amt) {

amount -= amt;

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public float getBalance() {

return amount;

}

}

/\*\*

\* 账户操作类

\*/

class AccountOperator implements Runnable{

private Account account;

public AccountOperator(Account account) {

this.account = account;

}

public void run() {

synchronized (account) {

account.deposit(500);

account.withdraw(500);

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

}

}

}

调用代码:

Account account = new Account("zhang san", 10000.0f);

AccountOperator accountOperator = new AccountOperator(account);

final int THREAD\_NUM = 5;

Thread threads[] = new Thread[THREAD\_NUM];for (int i = 0; i < THREAD\_NUM; i ++) {

threads[i] = new Thread(accountOperator, "Thread" + i);

threads[i].start();

}

结果如下：

Thread3:10000.0   
Thread2:10000.0   
Thread1:10000.0   
Thread4:10000.0   
Thread0:10000.0

在AccountOperator 类中的run方法里，我们用synchronized 给account对象加了锁。这时，当一个线程访问account对象时，其他试图访问account对象的线程将会阻塞，直到该线程访问account对象结束。也就是说谁拿到那个锁谁就可以运行它所控制的那段代码。   
当有一个明确的对象作为锁时，就可以用类似下面这样的方式写程序。

public void method3(SomeObject obj)

{

//obj 锁定的对象

synchronized(obj)

{

// todo

}

}

当没有明确的对象作为锁，只是想让一段代码同步时，可以创建一个特殊的对象来充当锁：

class Test implements Runnable

{

private byte[] lock = new byte[0]; // 特殊的instance变量

public void method()

{

synchronized(lock) {

// todo 同步代码块

}

}

public void run() {

}

}

说明：零长度的byte数组对象创建起来将比任何对象都经济――查看编译后的字节码：生成零长度的byte[]对象只需3条操作码，而Object lock = new Object()则需要7行操作码。

# **修饰一个方法**

Synchronized修饰一个方法很简单，就是在方法的前面加synchronized，public synchronized void method(){//todo}; synchronized修饰方法和修饰一个代码块类似，只是作用范围不一样，修饰代码块是大括号括起来的范围，而修饰方法范围是整个函数。如将【Demo1】中的run方法改成如下的方式，实现的效果一样。

****\*【Demo4】：synchronized修饰一个方法****

public synchronized void run() {

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

try {

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

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Synchronized作用于整个方法的写法。   
写法一：

public synchronized void method(){

// todo

}

写法二：

public void method(){

synchronized(this) {

// todo

}

}

写法一修饰的是一个方法，写法二修饰的是一个代码块，但写法一与写法二是等价的，都是锁定了整个方法时的内容。

在用synchronized修饰方法时要注意以下几点：   
1. synchronized关键字不能继承。   
虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。如果在父类中的某个方法使用了synchronized关键字，而在子类中覆盖了这个方法，在子类中的这个方法默认情况下并不是同步的，而必须显式地在子类的这个方法中加上synchronized关键字才可以。当然，还可以在子类方法中调用父类中相应的方法，这样虽然子类中的方法不是同步的，但子类调用了父类的同步方法，因此，子类的方法也就相当于同步了。这两种方式的例子代码如下：   
在子类方法中加上synchronized关键字

class Parent {

public synchronized void method() { }

}class Child extends Parent {

public synchronized void method() { }

}

在子类方法中调用父类的同步方法

class Parent {

public synchronized void method() { }

}class Child extends Parent {

public void method() { super.method(); }

}

1. 在定义接口方法时不能使用synchronized关键字。
2. 构造方法不能使用synchronized关键字，但可以使用synchronized代码块来进行同步。

# **修饰一个静态的方法**

Synchronized也可修饰一个静态方法，用法如下：

public synchronized static void method() {

// todo

}

我们知道静态方法是属于类的而不属于对象的。同样的，synchronized修饰的静态方法锁定的是这个类的所有对象。我们对Demo1进行一些修改如下：

****【Demo5】：synchronized修饰静态方法****

/\*\*

\* 同步线程

\*/

class SyncThread implements Runnable {

private static int count;

public SyncThread() {

count = 0;

}

public synchronized static void method() {

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

try {

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

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public synchronized void run() {

method();

}

}

调用代码:

SyncThread syncThread1 = new SyncThread();

SyncThread syncThread2 = new SyncThread();Thread thread1 = new Thread(syncThread1, "SyncThread1");Thread thread2 = new Thread(syncThread2, "SyncThread2");

thread1.start();

thread2.start();

结果如下：

SyncThread1:0   
SyncThread1:1   
SyncThread1:2   
SyncThread1:3   
SyncThread1:4   
SyncThread2:5   
SyncThread2:6   
SyncThread2:7   
SyncThread2:8   
SyncThread2:9

syncThread1和syncThread2是SyncThread的两个对象，但在thread1和thread2并发执行时却保持了线程同步。这是因为run中调用了静态方法method，而静态方法是属于类的，所以syncThread1和syncThread2相当于用了同一把锁。这与Demo1是不同的。

# **修饰一个类**

Synchronized还可作用于一个类，用法如下：

class ClassName {

public void method() {

synchronized(ClassName.class) {

// todo

}

}

}

我们把Demo5再作一些修改。   
****【Demo6】:修饰一个类****

/\*\*

\* 同步线程

\*/

class SyncThread implements Runnable {

private static int count;

public SyncThread() {

count = 0;

}

public static void method() {

synchronized(SyncThread.class) {

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

try {

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

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public synchronized void run() {

method();

}

}

其效果和【Demo5】是一样的，synchronized作用于一个类T时，是给这个类T加锁，T的所有对象用的是同一把锁。

内存溢出及解决方案

<http://blog.csdn.net/xianmiao2009/article/details/49254391>

Jvm内存模型

<https://www.cnblogs.com/xing901022/p/7725961.html>

Struct2跟spring mvc比较

1. strct2主要作用是在mvc的表现层，spring mvc是spring的一个组件，依靠spring实现mvc的一站式服务