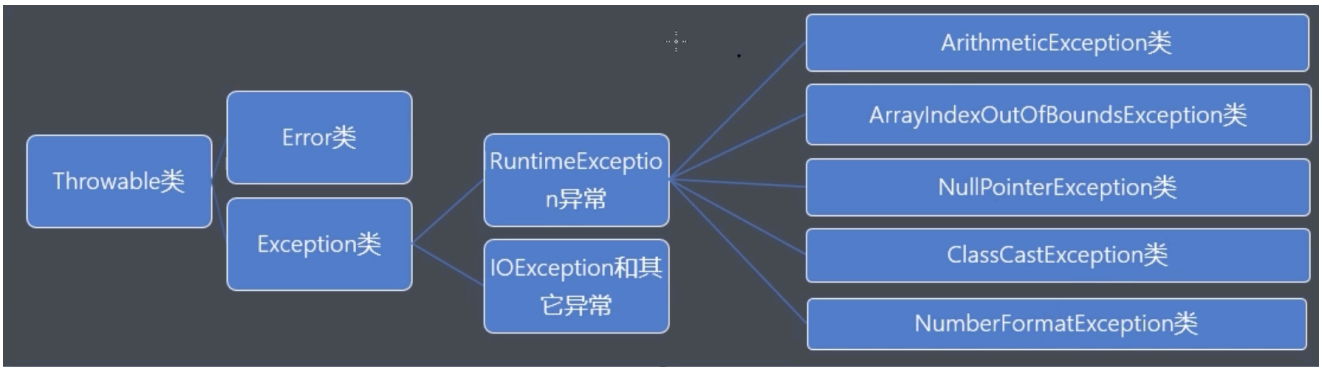
4.1 异常机制和File类

4.1.1 异常机制的基本概念

1. 异常就是不正常的含义，在Java语言中主要指程序执行中发生的不正常情况。
2. java.lang.Throwable类java语言中错误(Error)和异常(Exception)的超类。
3. Error类主要用于描述java虚拟机无法解决的严重错误，通常无法编码解决，如：JVM挂掉了
4. Exception类主要描述因编程错误或偶然外在因素导致的轻微错误，通常可以编码解决，如：0作为除数

4.1.2 异常机制的分类和结构

1. java.lang.Exception类是所有异常的超类，主要分为以下两种：
    - RuntimeException - 运行时异常，也叫作非检测异常
    - IOException和其他异常 - 其他异常，也叫作检测异常，所谓检测异常就是在编译阶段能被编译器检测出来的异常
2. RuntimeException类的主要子类：
    - ArithmaticException
    - ArrayIndexOutOfBoundsException
    - NullPointerException
    - ClassCastException
    - NumberFormatException
3. 当出现异常时，没有手动处理的话，java虚拟机采用默认方式处理异常，即打印异常的名称，原因，发生的位置以及终止程序。



4.1.4 异常的捕获实现

```
try{
  //执行此段代码，尝试捕获异常。编写有可能发生异常的代码
}
catch{
  //编写针对该类异常的处理代码
}
...
finally{
  //编写无论是否发生异常都要执行的代码
}
```

手动处理异常和没有手动处理的区别：代码是否可以继续向下执行。

4.1.6 finally的使用和笔试考点

finally通常进行善后处理，如：关闭已经打开的文件等。

4.1.7 异常的抛出

1. 子类不能抛出比父类更大的异常，或者平级不一样的异常。
2. 子类可以抛出与父类一样的异常
3. 子类可以抛出比父类更小的异常
4. 子类可以不抛出异常

4.1.9 自定义异常类的实现

1. 自定义xxxException异常类继承Exception类或者子类
2. 提供两个版本的构造方法，一个是无参构造，一个是以字符串作为参数的构造方法
3. 序列化的版本号，与序列化操作有关系

4.1.10 自定义异常类的使用

4.1.11 File类的概念和文件操作

java.io.File类主要描述**文件或目录**路径的抽象表示信息，可以**获取文件或目录的特征信息**，如，大小等。

方法声明	功能概述
File(String pathname)	
boolean exists()	
String getName()	
long length()	
long lastModified()	
String getAbsolutePath()	
boolean delete()	只能删除空目录
boolean createNewFile()	

4.1.12 File类实现目录操作

方法声明	功能概述
boolean mkdir()	创建单层目录
boolean mkdirs()	创建多层目录

4.1.13 File类实现目录的遍历

方法声明	功能概述
File[] listFiles()	获取该目录下所有的内容
boolean isFile()	
boolean isDirectory()	
File[] listFiles(FileFilter filter)	

匿名内部类的语法格式：接口/父类类型 引用变量名 = new 接口/父类类型() { 方法的重写 }

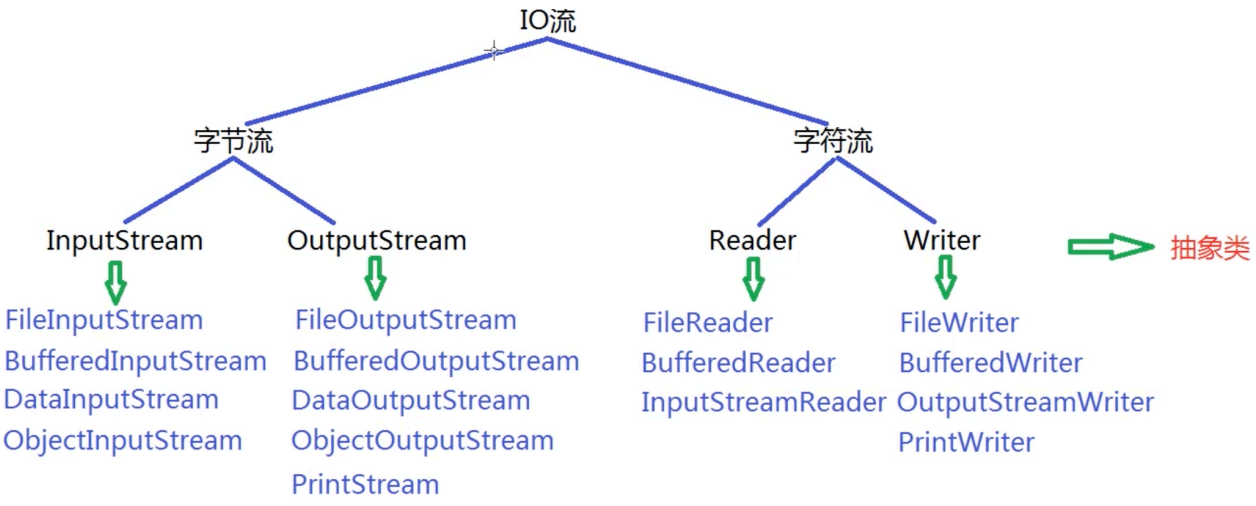
4.2 IO流

4.2.1 IO流的概念和分类

1. 按照读写数据的基本单位不同，分为字节流和字符流
    - 字节流主要以字节为单位进行数据读写的流，可以读写任意类型的文件
    - 字符流主要以字符(2个字节)为单位进行数据读写的流，只能读写文本文件。一个汉字两个字节
2. 按照读写数据的方向不同，可以分输入流和输出流(**站在程序的角度上**)
    - 输入流指从文件中读取数据内容输入到程序中，也就是读文件
    - 输出流指将程序中的数据内容输出到文件当中，也就是写文件
3. 按照流的角色不同分为节点流和处理流
    - 节点流主要指直接和**输入输出源**对接的流，直接和文件对接
    - 处理流主要指需要建立在节点流的基础之上的流，间接和文件对接

4.2.2 IO流的框架结构

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
访问文件	FileInputStream	FileOutputStream	FileReader	FileWriter
访问数组	ByteArrayInputStream	ByteArrayOutpuStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
访问字符串	--	--	StringReader	StringWriter
缓冲流	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
转换流	--	--	InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream	--	--
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流	--	PrintStream	--	PrintWriter
推回输入流	PushbackInputStream	--	PushbackReader	--
特殊流	DataInputStream	DataOutputStream	--	--



4.2.3 FileWriter类的概念和基本使用

java.io.FileWriter类主要用于将文本内容写入文本文件中。

方法声明	功能概述
FileWriter(String fileName)	**若文件不存在，该流会自动创建文件**；若文件存在，该流会清空文件中原有内容。
FileWriter(String fileName, boolean append)	以追加的方式创建对象去关联文件
void write(int c)	写入一个字符
void write(char[] cbuf, int offset, int length)	写入cbuf数组中从offset开始长度为length个字符
void write(char[] cbuf)	写入cbuf数组中所有字符
void flush()	
void close()	

4.2.5 FileReader类的概念和基本使用

java.io.FileReader类主要用于从文本文件中读取文本数据内容

方法声明	功能概述
FileReader()	
int read()	读取单个字符的数据并返回ASCII值，返回-1表示读到末尾
int read(char[] cbuf, int offset, int length)	从偏移量offset开始读取个数为length的字符到cbuf数组中，返回读取到字符的个数， 返回-1表示读到末尾
int read(char[] cbuf)	读取到整个cbuf数组，返回读取到字符的个数， 返回-1表示读到末尾
void close()	

4.2.7 文件字符流实现文件的拷贝

只能拷贝文本文件，想要拷贝其他格式的文件要用字节流

4.2.8 文件字节流实现文件的拷贝

方法声明	功能概述
FileOutputStream(String name)	
FileOutputStram(String name, boolean append)	
void write(int b)	写入单个字节
void write(byte[] bytes, int offset, int length)	写入bytes数组中从偏移量offset开始的length个字节
void write(byte[] bytes)	写入bytes数组中的所有字节
void flush()	
void close()	

方法声明	功能概述
FileInputStream(String name)	
int read()	读取单个字节的数据并返回ASCII值，返回-1表示读到末尾
int read(byte[] bytes, int offset, int length)	从偏移量offset开始读取length个字节到bytes数组中，返回读取到字符的个数， 返回-1表示读到末尾
int read(byte[] bytes)	读取到整个bytes数组中，返回读取到字符的个数， 返回-1表示读到末尾
void close()	
int available()	获取输入流所关联文件的大小

4.2.9 拷贝文件方式一的缺点

缺点：从硬盘中读取一个字节，拷贝一个字节，效率十分低。

4.2.10 拷贝文件方式二的实现和缺点

1. 先获取文件的大小

```
FileInputStream fis = new FileInputStream("d:/原视频.mp4");
FileOutputStream fos = new FileOutputStram("d:/拷贝视频.mp4");
int length = fis.available();
```

2. 准备一个大小一样的缓冲区，一次性的将文件的所有内容取到缓冲区然后一次性写入

```
byte[] arr = new Byte[length];
fis.read(arr);
fos.write(arr);
```

缺点：若文件过大时，无法申请和文件大小一样的缓冲区，内存不足

4.2.11 拷贝文件方式三

准备一个适当大小的缓冲区，1024的倍数。

```
byte[] arr = new Byte[1024];
int res = 0;
while((res = fis.read(arr)) != -1){
  fos.write(arr, 0, res);
}
```

4.2.12 缓冲字节流实现文件的拷贝

减少磁盘的交互次数，不用每个字节都和磁盘交互。

方法声明	功能概述
BufferedOutputStream(OutputStream out)	
BufferedOutputStream(OutputStream out, int size)	
void write(int b)	
void write(byte[] bytes, int offset, int length)	
void write(byte[] bytes)	
void flush()	
void close()	

方法声明	功能概述
BufferedInputStream(inputStream in)	
BufferedInputStream(inputStream in, int size)	
int read()	
int read(byte[] bytes, int offset, int length)	
Int read(byte[] bytes)	
void close()	
int available()	获取输入流所关联文件的大小

4.2.14 缓冲字符流的使用

BufferedReader

BufferedWriter

4.2.15

4.2.16

4.2.18 数据流的概念和使用

java.io.DataOutputStream类主要以适当的方式将基本数据类型写入输出流中

方法声明	功能概述
DataOutputStream(OutputStream out)	
void writeInt()	
void close()	

java.io.DataInputStream类主要从输出流中读取基本数据类型的数据

方法声明	功能概述
DataInputStream(InputStream in)	
void readInt()	
void close()	

4.2.19 ObjectOutputStream类的概念和使用

1. java.io.ObjectOutputStream类主要用于将一个对象的所有内容写入输出流当中
2. 只能将支持java.io.Serializable接口的对象写入流中
3. 类可以通过实现java.io.Serializable接口启动其序列化功能(需要加版本号)
4. 所谓序列化主要指将一个对象需要存储的相关信息有效组织成**字节序列**的转化过程

方法声明	功能概述
ObjectOutputStream(OutputStream out)	
void writeObject(Object obj)	
void close()	

4.2.20 ObjectInputStream类的概念和使用

方法声明	功能概述
ObjectInputStream(InputStream in)	
void readObject()	无法通过返回值判断是否读取到文件的末尾
void close()	

transient关键字修饰的变量不参加序列化，就是不会被写到文件中。

经验分享：当希望将多个对象写入文件当中，通常建议将多个对象放入到一个集合中，然后将这个集合看作一个对象写入输出流中，此时只需要调用一次readObject方法。

4.2.21RandomAccessFile类的概念和使用

java.io.RandomAccessFile类主要支持对随机访问文件的读写操作

方法声明	功能概述
RandomAccessFile(String name, String mode)	r: 只读； rw: 读写；rwd：读写，同步文件内容的更新；rws：读写，同步文件内容和元数据的更新
int read()	
void seek(long pos)	
void write(int b)	覆盖当前字符
void close()	

4.3 多线程

4.3.1 程序和进程的概念

1. 程序 -- 数据结构 + 算法，主要指**存放在硬盘上**的可执行文件
2. 进程 -- 主要指**运行在内存中**的可执行文件；一个运行中的程序叫做进程。是系统进行资源分配和调度的一个独立单元。
3. 目前主流的操作系统都支持多进程；但进程是重量级的，也就是新建一个进程会消耗CPU和内存空间等系统资源。

4.3.2 线程的概念和执行原理

1. 线程是进程内部的程序流，也就是说操作系统内部支持多进程的，而每个进程的内部又支持多线程。线程是轻量的，所有线程共享所在进程的所有资源。
2. 多线程是采用**时间片轮转法**来保证多个线程的并发执行，所谓并发就是宏观并行微观串行的机制。

4.3.3 线程的创建方式和相关方法

java.lang.Thread类代表线程，任何线程对象都是Thread类（子类）的实例。

两种创建线程的方式：

1. 自定义类继承Thread类并重写run方法，然后创建该类的对象调用start方法。
2. 自定义类实现Runnbale接口并重写run方法，创建该类的对象作为实参来构建Thread类型的对象，然后使用Thread类型的对象调用start方法。

方法声明	功能介绍
Thread()	
Thread(String name)	
Thread(Runnable target)	Runnbale是接口类型
Thread(Runnbale target, String name)	
void run()	若使用Runnable引用创建线程对象，调用该方法时最终调用接口中的版本；若没有使用Runnbale引用创建线程对象，调用该方法时啥也不做
void start()	用于线程启动，java虚拟机会自动调用该线程的run方法

4.3.5 线程创建和启动的方式一

调用run方法，本质上就是相当于对普通成员方法的调用，因此执行流程就是run方法的代码执行完毕后才继续向下执行。

调用start方法，相当于又启动一个线程。

4.3.6 线程创建和启动的方式二

4.3.7 匿名内部类的方式实现线程创建和启动

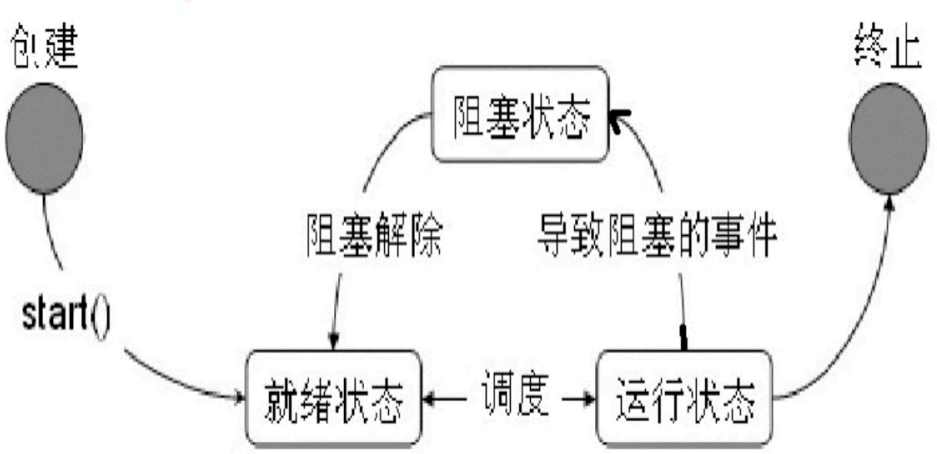
匿名内部类的语法格式：父类/接口了类型 引用变量名 = new 父类/接口类型() { 方法的重写 }

```
//方式一
Thread t1 = new Thread(){
  @Override
  public void run(){

  }
}

t1.start();

//方式二
Runnbale runnbale = new Runnbale(){
  @Override
  public void run(){

  }
}

Thread t2 = new Thread(runnable);
t2.start();
```

4.3.8 线程的生命周期

创建线程（利用new关键字把线程创建出来）--> start() --> **就绪状态**（此时线程还没有开始执行）<--> **运行状态**（线程调度器调用线程后进入的状态，当时间片执行完毕后任务没有完成时回到就绪状态）-->**消亡状态**（当线程的任务执行完后进入的状态，此时线程已经终止）

阻塞状态：当线程执行的过程中发生了阻塞事件进入的状态，如：sleep方法。阻塞状态解除后进入就绪状态。



4.3.9 继承方式管理线程编号和名称

方法声明	功能介绍
long getId()	相当于身份证号，不能修改
String getName()	
void setName()	
static Thread currentThread()	获取当前正在执行线程的引用

4.3.11 sleep方法的使用

方法声明	功能介绍
static void yield()	使当前线程让出处理器（离开Running状态），使当前线程进入Runnable状态等待
static void sleep(times)	使当前线程从Running放弃处理器进入Block状态，休眠times毫秒，再返回到Runnbale。如果其他线程打断当前线程的Block（sleep），就会方法IntteruptedException

4.3.12 线程优先级的管理

方法声明	功能介绍
int getPriority()	
void setPriority(int newPriority)	优先级越高不一定先执行，但该线程获取到时间片的机会会更多一些

4.3.13 线程的等待

方法声明	功能介绍
void join()	等待该线程终止
void join(long miles)	

4.3.14 守护线程

方法声明	功能介绍
boolean isDaemon()	
Void setDaemon(boolean on)	

对于新建的线程不是守护线程。

当子线程不是守护线程时，虽然主线程提前结束，但是子线程依然会继续执行，直到执行完毕。

当子线程是守护线程时，当主线程结束，子线程随之结束。

4.3.16 线程同步机制的概念和由来

1. 当多个线程同时访问一种共享资源时，可以会出现抢占资源的问题，此时就需要对线程之间进行**通信和协调**，该机制就叫做线程的同步机制。
2. 多个线程并发读写同一个临界资源时会发生线程并发安全问题。

4.3.17 同步代码块实现线程同步的方式一

synchronized关键字来实现同步锁机制从而保持线程执行的原子性（不可切割的最小单位）。所有对象都可以上同步锁。

两种上锁方式实现：

1. 使用同步代码块的方式实现**部分代码的锁定**，格式如下

```
synchronized(类类型的引用){ //注意引用必须是唯一的，相当于这部分资源的唯一一把锁
  //编写所有需要锁定的代码
}
```

4.3.19 同步方法实现线程同步的方式一

2. 使用同步方法的方式实现**所有代码**的锁定。直接使用synchronized关键字修饰整个方法即可。

```
//等价
synchronized(this){
  //编写所有需要锁定的代码
}
```

4.3.20 同步方法实现线程同步的方式二

1. 当我们对一个静态方法加锁，如：public synchronized static void xxx(){ .... }
2. 那么该方法锁的对象是类对象。每个类都有唯一的一个类对象。获取类对象的方式：**类名.class**

4.3.21线程安全和死锁的问题

使用synchronizd保证线程同步应当注意：

1. 使用同步代码块时，必须是同一个对象
2. 尽量减少同步范围

StringBuffer类是线程安全的类；StringBuilder类不是线程安全的。

ArrayList类和HashMap类不是线程安全的类，但可以用Collections.synchronizedList() 和 Collections.synchronizedMap()等方法实现安全。

4.3.22 使用Lock锁实现线程同步

1. 从java5开始新增的功能
2. java.util.concurrent.Locks接口是控制多个线程对共享资源进行访问的工具。
3. 该接口主要实现类是ReentrantLock类，该类拥有与synchronized相同的并发性，在以后的线程安全控制中，经常使用ReentrantLock类显示加锁和释放锁。

方法声明	功能介绍
ReentrantLock()	
void lock()	
Void unlock()	

与synchronized方式的比较：

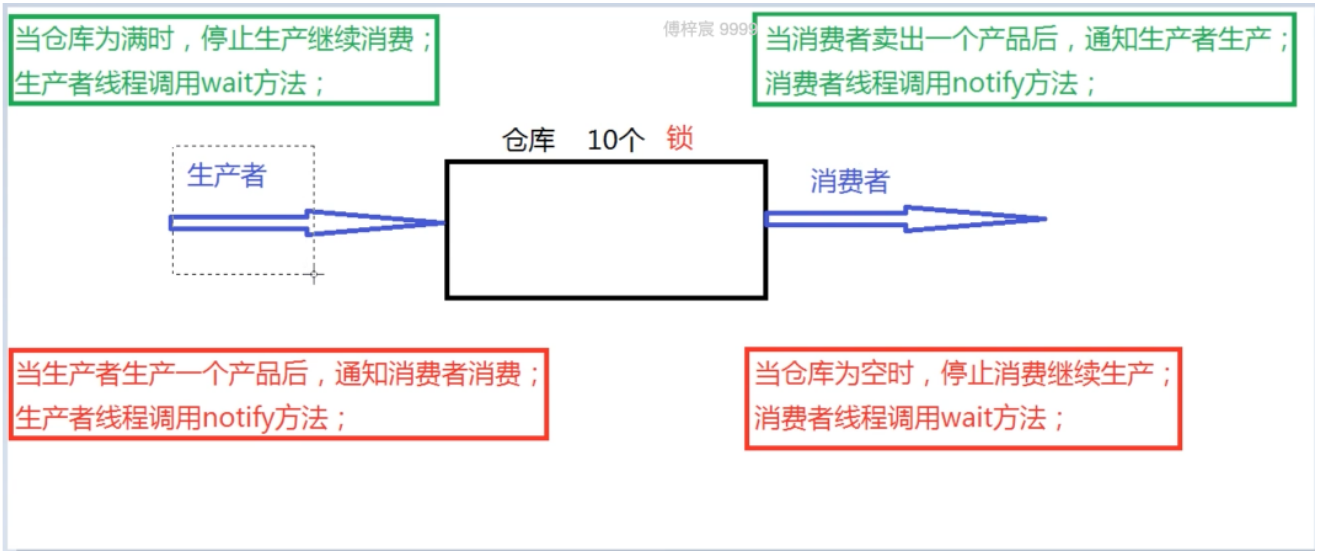
1. lock是显示锁，需要手动实现开启和关闭操作，而synchronized是隐式锁，执行锁定代码后自动释放。
2. lock只有同步代码块方式的锁，而synchronized有同步代码块和同步方法两种锁。
3. 使用lock锁方式时，性能更好。

4.3.23 线程之间的通信

方法声明	功能介绍
void wait()	用于让线程进入阻塞状态，自动释放锁。直到其他线程调用notify()或notifyAll()方法。
void wait(long timeout)	
void notify()	唤醒单个线程
void notifyAll()	唤醒所有线程

必须在锁定的代码中调用。

4.3.24 生产者消费者模型的概念



4.3.25 生产者消费者模型的实现

```
public class StoreHouse(){

  private int cnt = 0;

  public synchronized void produceProduct(){
    notify();
    if(cnt < 10){
      System.out.println("线程" + Thread.currentThread().getName() + "正在生产第" +
(cnt+1) + "个产品...");
      cnt++;
    }else{
```

```
      try{
        wait();
      }catch(InterruptedException e){
        e.printStrachTrace();
      }
    }
  }

   public synchronized void consumerProduct(){
     notify();
     if(cnt > 0){
       System.out.println("线程" + Thread.currentThread().getName() + "消费第" + cnt
+ "个产品...");
       cnt--;
     }else{
        try{
          wait();
        }catch(InterruptedException e){
          e.printStrachTrace();
        }
     }
   }

}
```

```
public class ProduceThread extends Thread{
  private StoreHouse storeHouse;

  public ProduceThread(StoreHouse storeHouse){
    this.storeHouse = storeHouse;
  }

  @Override
  public void run(){
    while(true){
      storeHouse.produceProduct();
      try{
        Thread.sleep(100);
      }catch(){
        e.printStackTrace();
      }
    }
  }
}
```

```
public class ConsumerThread extends Thread{
  private StoreHouse storeHouse;

  public ConsumerThread(StoreHouse storeHouse){
    this.storeHouse = storeHouse;
  }

  @Override
  public void run(){
    while(true){
      storeHouse.consumerProduct();
      try{
        Thread.sleep(1000);
      }catch{
        e.printStackTrace();
      }
    }
  }
}
```

```
public class Test{
  public static void main(String[] args){
    StoreHouse storeHouse = new StoreHouse();

    ProduceThread t1 = new ProduceThread(storeHouse);
    ConsumerThread t2 = new ConsumerThread(storeHouse);
    t1.start();
    t2.start();
  }
}
```

4.3.26 创建和启动线程的方式三

从java5开始，可以通过实现java.util.concurrent.Callable接口

方法声明	功能介绍
V call()	计算并返回结果

java.util.concurrent.FutureTask类

方法声明	功能介绍
FutureTask(Callable callable)	
V get()	用来获取call方法计算结果

4.3.27 线程池的概念和使用

首先创建一些线程，它们的集合称为线程池，当服务器接受到一个客户请求后，就从线程池中取出一个空闲的线程为之服务，服务完后不关闭线程，而是将该线程还会到线程池中。

在线程池的编程模式下，任务是交给整个线程池的，而不是直接交给某个线程，线程池拿到任务后，它就在内部找有无空闲的线程，再把任务交给某个空闲的线程。任务是交给整个线程池的，一个线程只能执行一个任务，但可以向线程池提交多个任务。

从java5开始提供了线程池相关的类和接口：java.util.concurrent.Executors类和java.util.concurrent.ExecutorService接口。

Executors类可以创建并返回不同类型的线程池

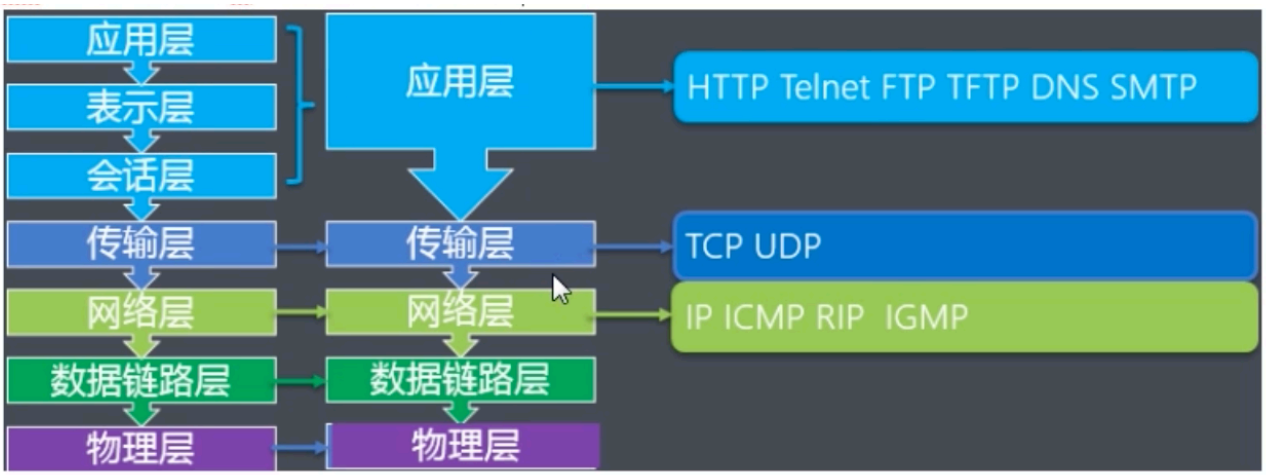
static ExecutorService newCachedThreadPool()	
static ExecutorService newFixedThreadPool(int nThreads)	
static ExecutorService newSingleThreadExecutor()	

其中ExecutorService接口是真正的线程池接口

void execute(Runnable command)	执行任务和命令，通常用于执行Runnable
Future submit(Callable task)	执行任务和命令，通常用于执行Callable
void shutdown()	

4.4 网络编程

4.4.1 七层网络模型



4.4.2 相关的协议

协议的概念

计算机之间的通信需要一些约定或者规则，这种约定和规则叫做通信协议。通信协议可以对速率，传输代码，代码结构，传输控制步骤，出错控制等制定统一的标准。

TCP协议

传输控制协议（Transmission Control Protocol），是面向连接的协议。

1. 建立连接 --> 进行通信 --> 断开链接
2. 在传输前采用“三次握手”的方式
3. 在通信的整个过程中全程保持连接，形成数据传输通道
4. 保证了数据传输的可靠性和有序性
5. 是一种全双工的字节流通信方式，可以进行大数据量的传输
6. 传输完毕后需要释放已建立的连接，采用“四次挥手”方式，发送数据的效率比较低

UDP协议

用户数据报协议（User Datagram Protocol），是一种非面向连接的协议。

1. 在通信的整个过程中不需要保持连接，其实是不需要建立连接
2. 不保证数据传输的可靠性和有序性
3. 是一种全双工的数据报通信方式，每个数据报的大小限制在64k内
4. 发送数据完毕不需要释放资源，开销小，发送数据的效率高，速度快

4.4.3 IP地址和端口号

IP地址

1. IP地址是互联网的唯一地址标识，本质上有32位二进制组成的整数，叫做IPv4. 当然也有128位组成的整数，叫做IPv6。目前主流还是IPv4。
2. 日常生活中采用**点分十进制**表示法来进行IP地址的描述，将每个字节的二进制转化为一个十进制整数，不同的整数之间采用小数点隔开。

3. IP地址可以找到具体某一台设备。

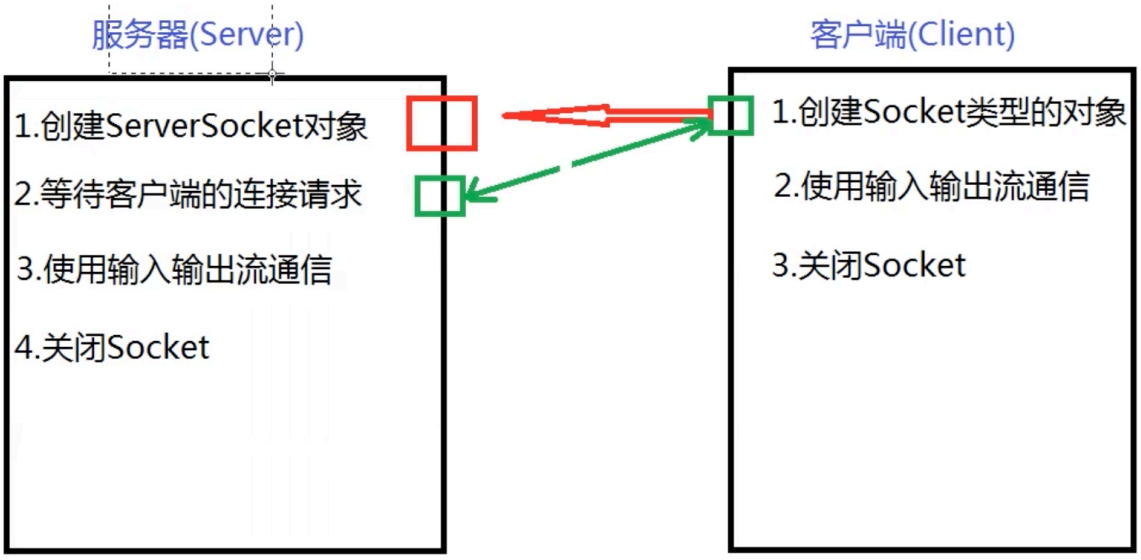
端口号

1. 端口号可以定位到该设备中具体某一进程。
2. 本质上是16位二进制组成的整数，其中0～1024之间的端口号通常被系统占用，建议从1025开始。

4.4.4 基于tcp协议的编程模型

tcp协议的编程模型 - 打电话

就是s/c架构，s/c架构下客户向服务器发出请求，服务器接受请求后提供服务



ServerSocket类

1. java.net.ServerSocket类主要用于描述服务端套接字信息。

方法声明	功能介绍
ServerSocket(int port)	
Socket accept()	侦听并接受连接请求
void close()	

Socket类

1. java.net.Socket类主要用于描述客户端套接字信息，是两台机器间通信的断点。

方法声明	功能介绍
Socket(String host, int port)	服务器的主机名和端口号
InputStream getInputStream()	
OutputStream getOutputStream()	
void close()	

4.4.5 基于tcp协议模型的框架实现

```
public class ServerStringTest {
  public static void main(String[] args){
    //创建ServerSocket类型的对象并提供端口号
    ServerSocket ss = new ServerSocket(8888);

    //等待客户端的连接请求，调用accept方法
    //当没有客户端连接时，服务器阻塞在accept方法的调用这里
    Socket s = ss.accept();

    //使用输入输出流进行通信

    //关闭资源
    s.close();
    ss.close();
  }
}
```

```
public class ClientStringTest {
  //创建Socket类型的对象并提供服务器的主机名和端口号
  Socket s = new Socket("127.0.0.01", 8888);

  //使用输入输出流进行通信

  //关闭资源
  s.close();

}
```

4.4.6 客户端向服务器放送数据的实现

```
public class ServerStringTest {
  public static void main(String[] args){
    //创建ServerSocket类型的对象并提供端口号
    ServerSocket ss = new ServerSocket(8888);

    //等待客户端的连接请求，调用accept方法
    //当没有客户端连接时，服务器阻塞在accept方法的调用这里
    Socket s = ss.accept();

    //使用输入输出流进行通信
    //实现对客户端发来字符串内容的接受并打印
    BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
    String s1 = br.readLine();

    //关闭资源
    br.close();
    s.close();
    ss.close();
  }
}
```

```
public class ClientStringTest {
  //创建Socket类型的对象并提供服务器的主机名和端口号
  Socket s = new Socket("127.0.0.01", 8888);

  //使用输入输出流进行通信
  //实现客户端向服务器发送字符串内容“hello”
  PrintStream ps = new PrintStream(s.getOutputStream());
  ps.println("hello");

  //关闭资源
  ps.close();
  s.close();
}
```

4.4.7 服务器向客户器放送数据的实现

```
public class ServerStringTest {
  public static void main(String[] args){
    //创建ServerSocket类型的对象并提供端口号
    ServerSocket ss = new ServerSocket(8888);

    //等待客户端的连接请求，调用accept方法
    //当没有客户端连接时，服务器阻塞在accept方法的调用这里
    Socket s = ss.accept();

    //使用输入输出流进行通信
    //实现对客户端发来字符串内容的接受并打印
    BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
    String s1 = br.readLine();
    //实现服务端向客户端回发
    PrintStream ps = new PrintStream(s.getOutputStream());
    ps.println("I received");

    //关闭资源
    ps.close();
    br.close();
    s.close();
    ss.close();
  }
}
```

```
public class ClientStringTest {
  //创建Socket类型的对象并提供服务器的主机名和端口号
  Socket s = new Socket("127.0.0.01", 8888);

  //使用输入输出流进行通信
  //发送的消息从键盘输入
  System.out.println("请输入想要发送的内容：");
  Scanner sc = new Scanner(System.in);
  String str1 = sc.next();
  PrintStream ps = new PrintStream(s.getOutputStream());
  ps.println(str1);
  //接受服务器发来的信息
  BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
  String str2 = br.readLine();
```

```
  //关闭资源
  br.close();
  ps.close();
  sc.close();
  s.close();
}
```

4.4.8 客户端和服务器不断通信的实现

4.4.9 服务器采用多线程机制的实现

每当有一个客户端连接成功，则需要启动一个新的线程。

```
public static ServerThread extends Thread{

  private Socket s;

  public ServerThread(Socket s){
    this.s = s;
  }

  @Override
  public void run(){
    Buffered br = null;
    PrintStream ps = null;

    try{
      br = new BufferedReader(new InputStream(s.getInputSteam()));
      ps = new PrintStream(s.getOutSteam());

      while(true){
        String s1 = br.readLine();
        System.out.println("客户端发来的字符内容是：" + s1);
        if(s1.equalsIgnoreCase("bye")){
          System.out.println("客户端已下线！");
          break;
        }

        ps.println("I received");
        System.out.println("服务器发送数据成功！");
      }
    }catch(IOException e){
      e.printStackTrace();
    }finally{
      if(ps != null){
```

```
        ps.close();
      }
      if(br != null){
        try{
          br.close();
        }catch(IOException e){
           e.printStackTrace();
        }
      }
      if(s != null){
         try{
          s.close();
        }catch(IOException e){
           e.printStackTrace();
        }
      }
    }
  }
}
```

```
public class ServerStringTest {
  public static void main(String[] args){
    ServerSocket ss = null;
    Socket s= null;
    try{
      //创建ServerSocket类型的对象并提供端口号
      ss = new ServerSocket(8888);
      while(true){
        System.out.println("等待客户端的连接请求...");
        //等待客户端的连接请求，调用accept方法
        //当没有客户端连接时，服务器阻塞在accept方法的调用这里
        s = ss.accept();
        System.out.println("客户端连接成功！");
        //每当有一个客户端连接成功，则需要启动一个新的线程。
        new ServerThread(s).start();
      }
    }catch(IOException e){
      e.printStackTrace();
    }finally{
      if(ss != null){
        try{
          ss.close();
        }catch(IOException e){
           e.printStackTrace();
        }
```

```
      }
    }
  }
}
```

```
public class ClientStringTest {

  public static void main(String[] args){
    Socket s = null;
    PrintStream ps = null;
    Scanner sc = null;
    Buffered br = null;

    try{
      //创建Socket类型的对象并提供服务器的主机名和端口号
      s = new Socket("127.0.0.01", 8888);
      System.out.println("连接服务器成功！");

      //使用输入输出流进行通信
      //发送的消息从键盘输入
      sc = new Scanner(System.in);
      ps = new PrintStream(s.getOutputStream());
      br = new BufferedReader(new InputStreamReader(s.getInputStream()));

      while(true){
        System.out.println("请输入想要发送的内容：");
        String str1 = sc.next();
        ps.println(str1);
        String str2 = br.readLine();
        System.out.println("服务器回发的字符内容是：" + str2);
      }
    }catch(){

    }finally{
      //关闭资源
      br.close();
      ps.close();
      sc.close();
      s.close();
    }
  }
}
```

4.4.10 基于udp协议的编程模型

编程模型 - 寄信

接收方：

1. 创建DatagramSocket类型的对象并提供端口号
2. 创建DatagramPacket类型的对象并提供缓冲区
3. 通过Socket接受数据内容存放到Packet中，调用receive方法
4. 关闭Socket

发送方：

1. 创建DatagramSocket类型的对象
2. 创建DatagramPacket类型的对象并提供接收方的通信地址
3. 通过Socket将Packet中的数据内容发送出去，调用send方法
4. 关闭Socket

DatagramSocket类

主要用于描述发送和接受数据报的套接字

方法声明	功能介绍
DatagramSocket()	
DatagramSocket(int port)	
void receive(DatagramPacket p)	
void send(DatagramPacket p)	
void close()	

DatagramPacket类

主要描述数据报

方法声明	功能介绍
DatagramPacket(byte[] buf, int length)	用于接受长度为length的数据报
DatagramPacket(byte[] buf, int length, InetAddress address, int port)	将数据报发送到指定地址和端口
InetAddress getAddress()	
int getPort()	
int getLength()	

InetAddress类

描述互联网通信地址信息

方法声明	功能介绍
static InetAddress getLocalHost()	
static InetAddress getByName(String host)	

4.4.13 URL类的概念和使用

基本概念

1. java.net.URL（Unifrom Resource Identifier）类主要用于表示统一的资源定位器。

常用方法

方法声明	功能介绍
URL(String spec)	
String getProtocol()	
String getHost()	
int getPort()	
String getPath()	
String getFile()	
URLConnection openConnection()	

URLConnection类

URLConnection类是个抽象类，该类表示应用程序和URL之间的通信链接的所有类的超类，主要实现类有支持HTTP特有功能的HttpURLConnection类。

方法声明	功能介绍
InputStream getInputStream()	
Void disconnect()	

4.5 反射机制

4.5.1 反射机制的基本概念

在某些特殊场合中编写代码时不确定要创建什么类型的对象，也不确定要调用什么样的方法，这些都希望通过运行时传递的参数来决定，该机制叫做动态编程技术，也就是反射机制。

通俗来说，反射机制就是用于动态创建对象并且动态调用方法的机制。

4.5.2 Class类的概念和Class对象的获取方式

基本概念

1. java.lang.Class类的实例可以用于描述java应用程序中的类和接口，也就是一种数据类型。
2. 没有公共构造方法，该类的实例是有java虚拟机和类加载器自动构造完成，本质上就是加载到内存中的运行时类

获取Class对象的方式

1. 使用数据类型.class的方式可以获取对应类型的Class对象（掌握）
2. 使用对象.getClass()的方式获取对应的Class对象
3. 使用包装类.TYPE的方式可以获取对应基本数据类型的Class对象
4. 调用Class类中的forName方法来获取对应的Class对象，要求写完整的名称：包名.类名（掌握）

```
c1 = Class.forName("String"); //error
c1 = Class.forName("java.lang.String");
```

5. 使用类加载器的方法来获取Class对象

```
ClassLoader classLoader = String.class.getClassLoader();
c1 = classLoader.loadClass("java.lang.String");
```

4.5.4 无参方式创建对象的两种形式

方法声明	功能介绍
T newInstance()	创建该Class对象所表示类的新实例

4.5.5 无参方式构造对象的优化

Class类的常用方法

方法声明	功能介绍
Constructor getConstructor(Class<?>... parameterTypes)	参数可变，参数必须时Class类型
Constructor<?>[] getConstructors()	获取所有构造方法

Constructor类

java.lang.reflect.Constructor类主要描述获取到的构造方法信息

方法声明	功能介绍
T newInstance(Object... initargs)	
int getModifiers()	
String getName()	
Class<?>[] getParameterTypes()	

4.5.8 获取成员变量数值的两种形式

Class类的常用方法

方法声明	功能介绍
Field getDeclaredField(String name)	获取成员变量信息
Filed getDeclaredFilelds()	

Filed类

java.lang.reflect.Filed类主要描述取到的单个成员变量信息

方法声明	功能介绍
Object get(Object obj)	获取参数对象obj中此Filed对象所表示成员变量的数值
void set(Object obj, Object value)	
void setAccessible(boolean flag)	
int getModifiers()	
Class<?> getType()	
String getName()	

4.5.11 获取成员方法的两种形式

Class类的常用方法

方法声明	功能介绍
Method getMethod(String name, Class<?>... parametersTypes)	
Method getMethods[]	

Method类

java.lang.reflect.Method类主要描述获取到的单个成员方法信息

方法声明	功能介绍
Object invoke(Object obj, Object... args)	
int getModifiers()	
Class<?> getReturnType()	
String getName()	
Class<?>[] getParameterTypes()	
Class<?>[] getExceptionTypes()	

4.5.13 获取其他结构的实现

4.6 作业

4.6.1 作业一

对于ObjectInputStream的EOFException该怎么解决?

4.6.5 作业五

1. 终止客户端后服务器端会有异常
2. 如何让各个线程拿到的最新的list
3. Bufferedwriter VS PrintStream