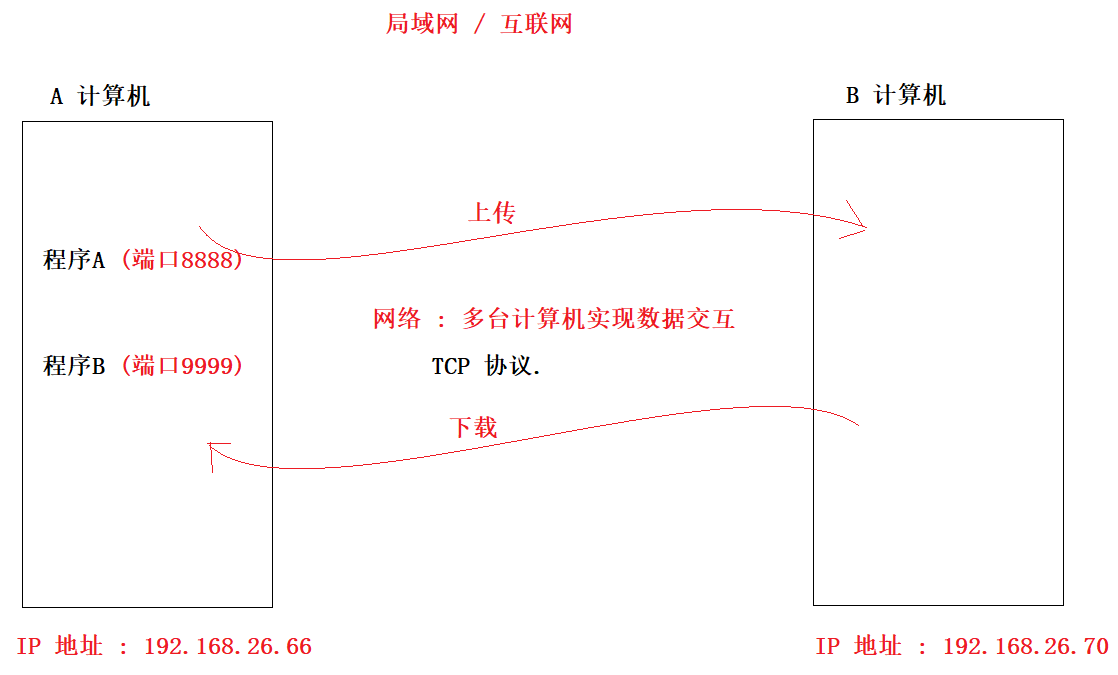
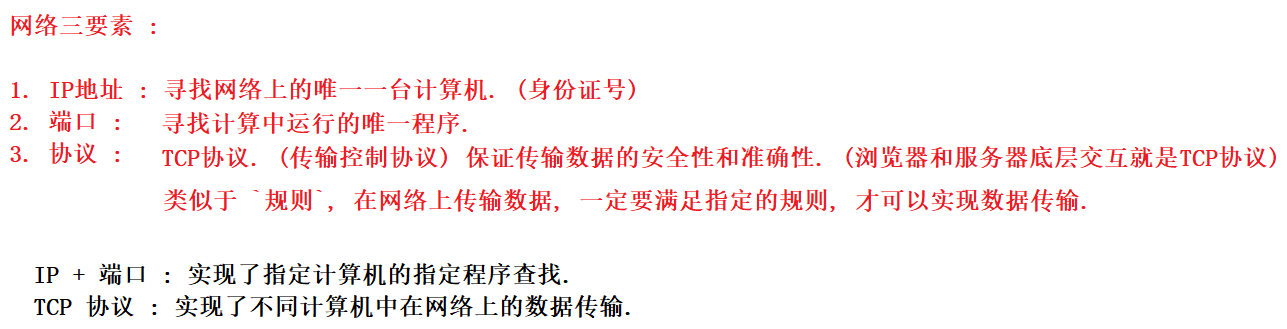
# 1. 网络编程 :

**网络 : 将互联网上的多台计算机实现数据交互.**

IO 流 : A计算机中的数据复制到A计算机.

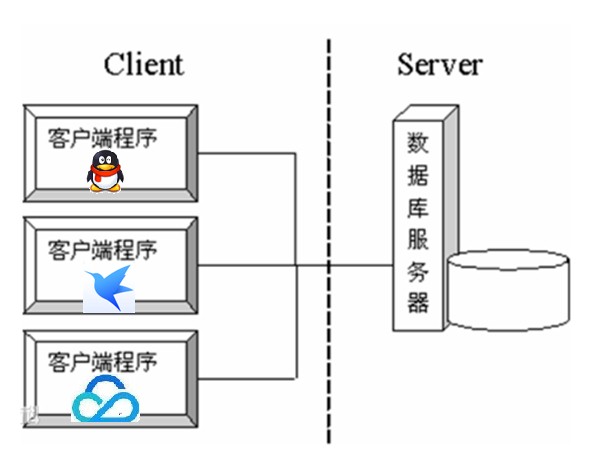
IO 流 + 网络 : A计算机中的数据复制到B计算机.





## 1. 软件结构 : (了解)

**C/S结构**：全称为Client/Server结构，是指客户端和服务器结构。常见程序有ＱＱ、迅雷等软件。



好处 :

1. 一些数据的处理可以直接在客户端实现. 减轻服务器的压力.

缺点 :

1. 公司需要两套开发人员, 一套为客户端开发人员. 从而会增加的运营成本.
2. 如果程序出现问题. 也需要两套人员共同去解决, 从而也提升了维护成本.
3. 用户在自己的设备上下载该客户端应用, 才可以使用.

B/S结构：全称为Browser/Server结构，是指浏览器和服务器结构。常见浏览器有谷歌、火狐, IE等。

好处 :

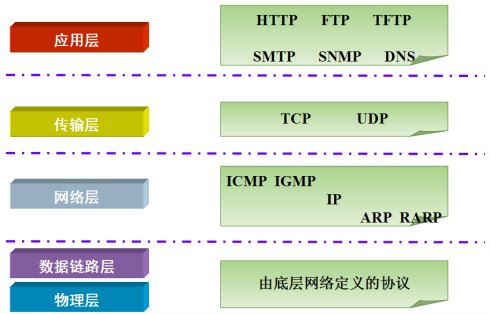
1. 浏览器不需要用户安装, 因为所有的操作系统都会自带浏览器.
2. 浏览器也不需要自己的公司开发, 操作系统自带.
3. 减轻了公司的开发成本与维护成本.

缺点 :

1, 浏览器只能实现提交数据, 因此所有的数据都在服务器实现处理, 从而增加了服务器的压力.

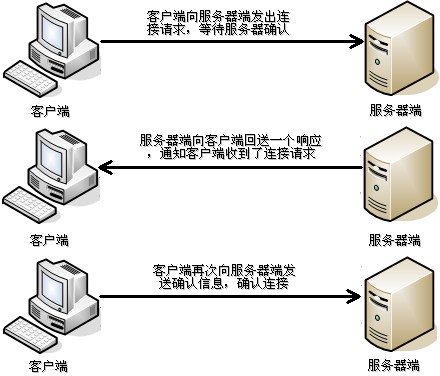
## 2. 网络通信协议 : (了解)

**网络的 TCP / IP 结构 : 源于 7层, 简化为4层.**

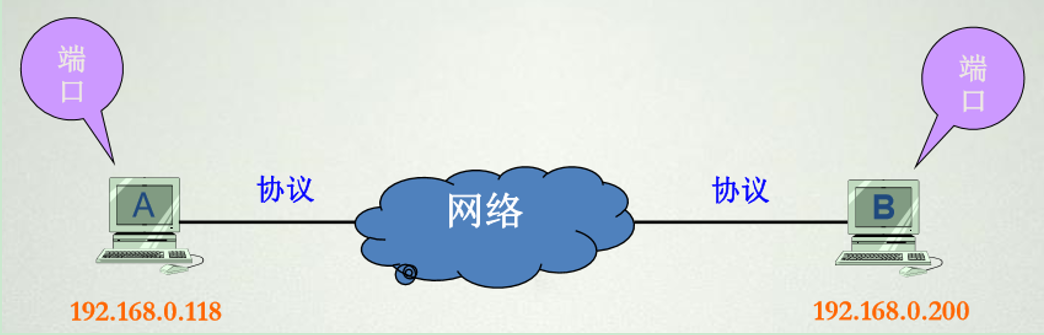


**TCP 协议的三次握手 : 目的, 确立连接, 一旦连接建立, 数据就可以在连接通道中实现传输. TCP协议会维护这条通道.**

**三次握手实现中, 请求 (Request) 必须由浏览器发送, 服务器不能主动发送. 服务器只能等待, 然后做出响应(Response).**



## 3. 网络编程三要素 : (了解)



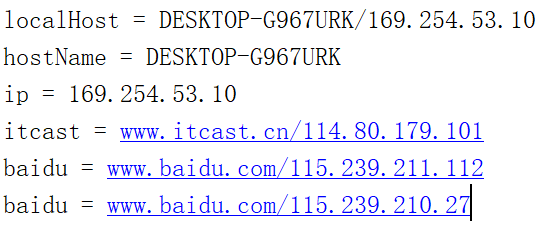
1. **Ip地址:**

Java中有一个类专门用来表示 IP 对象. IP对象中包含了 `主机名称和ip地址`. Ip对象所属的类名称为 InetAddress.

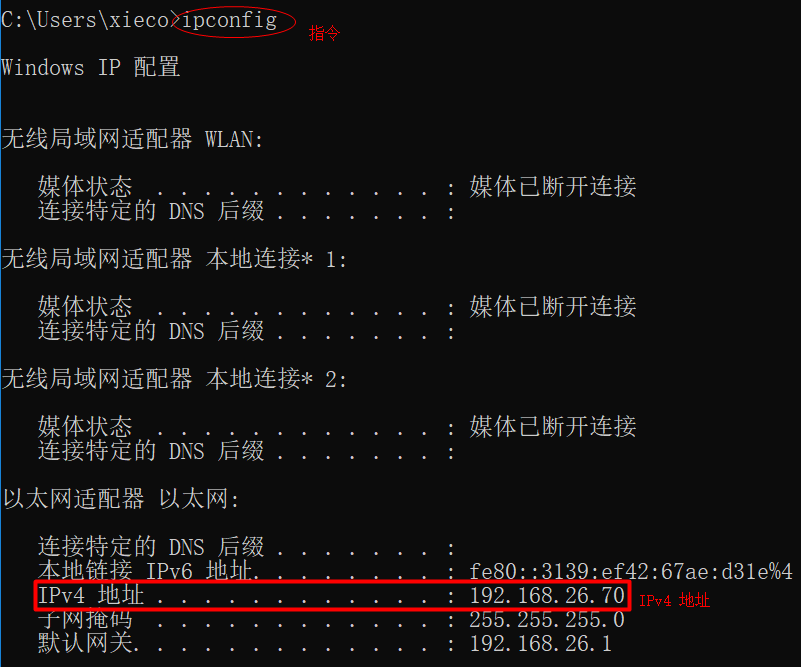
**类 InetAddress :** IP 地址是 IP 使用的 32 位(Ipv4) 或 128(Ipv6) 位无符号数字，它是一种低级协议，UDP 和 TCP 协议都是在它的基础上构建的。

|  |  |
| --- | --- |
| [String](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/lang/String.html) | [**getHostName**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/net/InetAddress.html#getHostName())()            获取此 IP 地址的主机名。 |
| [String](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/lang/String.html) | [**getHostAddress**](mk:@MSITStore:C:\Users\xieco\Desktop\JDK9%20API文档\_JDK6API中文参考%5b沈东良%5d(070114).chm::/java/net/InetAddress.html#getHostAddress())**()            返回 IP 地址字符串（以文本表现形式）。** |

public class Test1 {  
 public static void main(String[] args) throws UnknownHostException {  
  
 *// static InetAddress getLocalHost() 返回本地主机。(自己电脑)* InetAddress localHost = InetAddress.*getLocalHost*(); *// 地址: 127.0.0.1 `本机回环地址`  
 // localHost = DESKTOP-G967URK (主机名称) / 169.254.53.10 (外网地址)* System.*out*.println("localHost = " + localHost);  
  
 String hostName = localHost.getHostName();  
 String ip = localHost.getHostAddress();  
 System.*out*.println("hostName = " + hostName);  
 System.*out*.println("ip = " + ip);  
  
 *// static InetAddress getByName(String host) 在给定主机名的情况下确定主机的 IP 地址。* InetAddress itcast = InetAddress.*getByName*("www.itcast.cn");  
 System.*out*.println("itcast = " + itcast);  
  
 *// static InetAddress[] getAllByName(String host) 在给定主机名的情况下，根据系统上配置的名称服务返回其 IP 地址所组成的数组。* InetAddress[] baidus = InetAddress.*getAllByName*("www.baidu.com");  
 for (InetAddress baidu : baidus) {  
 System.*out*.println("baidu = " + baidu);  
 }  
 }  
}



**请问 :** 如何查询计算机的局域网地址 ??? 指令: ipconfig



1. **port 端口 :** 寻找指定计算中的指定程序. 端口号的范围 0 ~ 65535, 这是两个字节的最大长度.

注意点 : 端口号 0 ~ 1024 之间的端口号, 程序不要自己使用. 因为这些端口是预留给知名的服务商来使用. 比如: HTTP 80 HTTPS 443 …

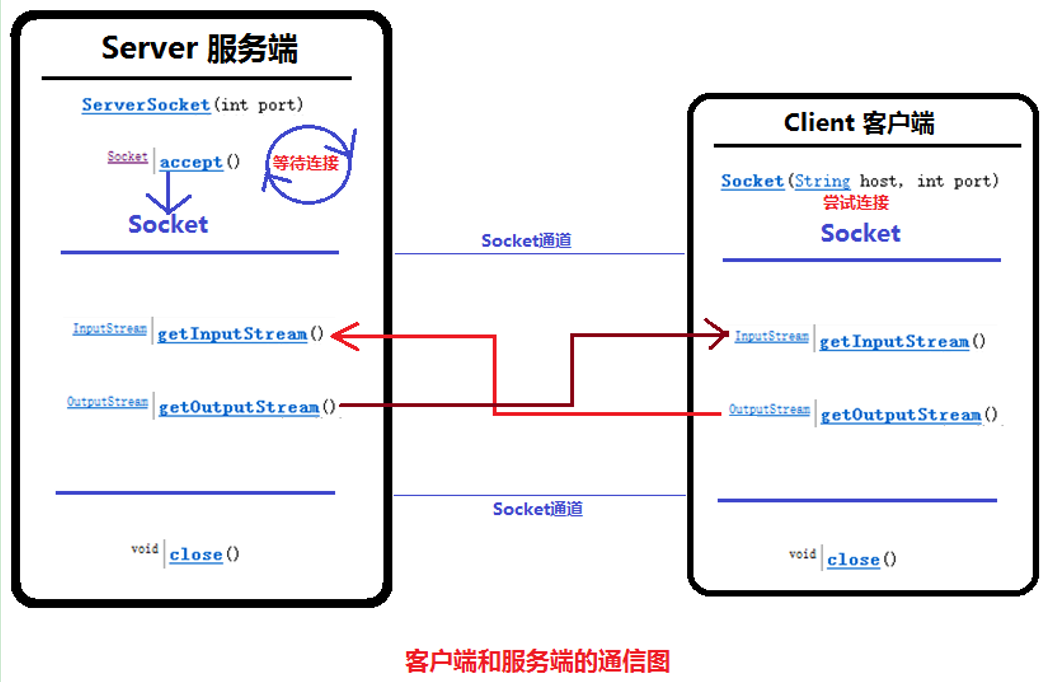
自己程序开辟端口, 尽量使用 1024 以上的端口.

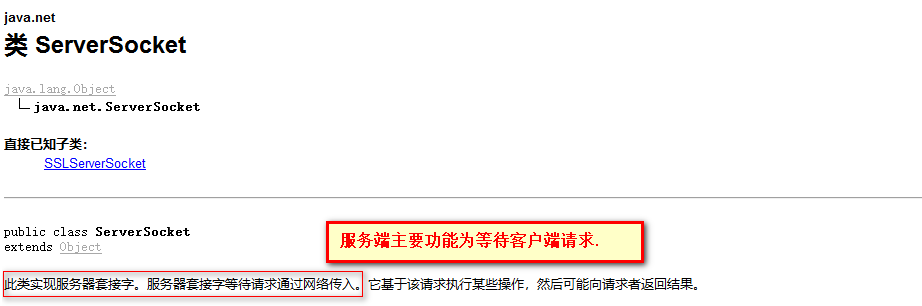
指令 : netstat -nao



1. **TCP协议 :** 帮助程序实现不同计算机在网络上的数据传输.

## 4. 简单的TCP程序 :

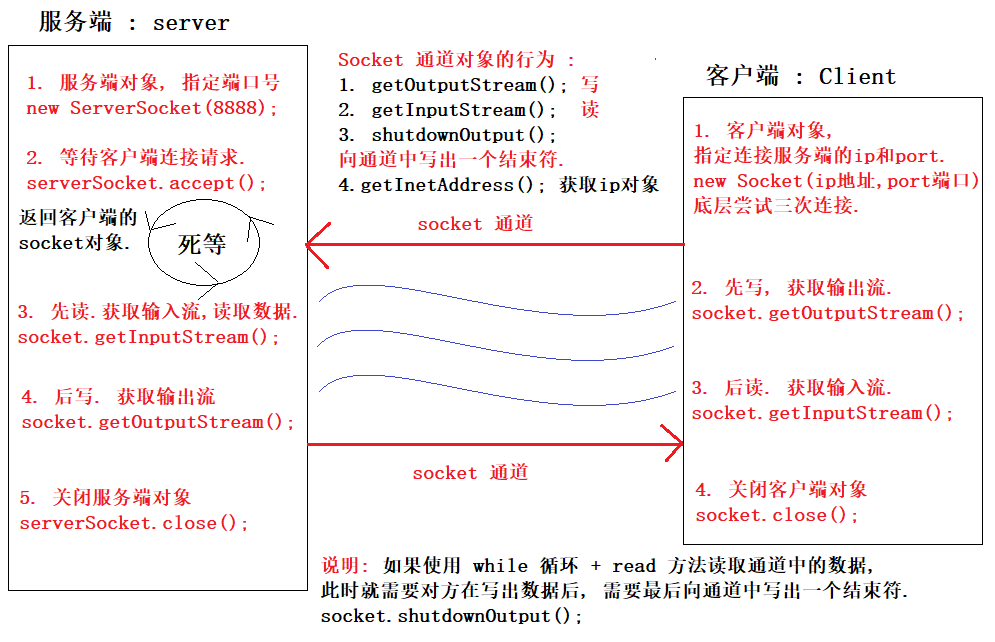






### 1. 客户端向服务器发送数据 :



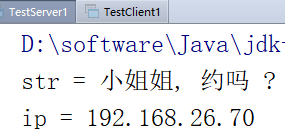


**服务端实现 :**

public class TestServer1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字 (ServerSocket)* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待客户端连接 (死等)* Socket socket = serverSocket.accept();  
  
 *// 3. 先读 (获取socket通道的输入流)* InputStream in = socket.getInputStream();  
 byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = in.read(buf)) != -1) {  
 *// out.write(buf, 0, len); 将数据写入到文件中.  
 // String 类的构造方法 : String(byte[] bytes, int offset, int length)* String str = new String(buf, 0, len);  
 System.*out*.println("str = " + str);  
 String ip = socket.getInetAddress().getHostAddress();  
 System.*out*.println("ip = " + ip);  
 }  
  
 *// 4. 关闭服务端套接字* serverSocket.close();  
 }  
}

**客户端实现 :**

public class TestClient1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个客户端套接字对象  
 // new Socket("localhost", 8888);  
 // new Socket("127.0.0.1", 8888);* Socket socket = new Socket("192.168.26.70", 8888); *// 尝试三次连接  
  
 // 2. 写入数据到通道中* OutputStream out = socket.getOutputStream();  
 out.write("小姐姐, 约吗 ?".getBytes());  
 out.flush();  
  
 *// 3. 关闭客户端套接字* socket.close();  
 }  
}



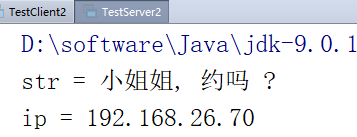
### 2. 服务器向客户端回送数据 :

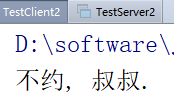
**服务端实现 :**

public class TestServer2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字 (ServerSocket)* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待客户端连接 (死等)* Socket socket = serverSocket.accept(); *// 阻塞方法.  
  
 // 3. 先读 (获取socket通道的输入流)* InputStream in = socket.getInputStream();  
 byte[] buf = new byte[1024];  
 int len = -1;  
 *// in.read(buf); read 方法也是一个阻塞方法.  
 // 文件读取, 读取到末尾时, 可以读取到 -1 数据. 结束循环.  
 // socket通道读取, 读取不到 -1, 循环会卡死, 此时需要对方调用 socket 对象的 shutdownOutput 方法.* while ((len = in.read(buf)) != -1) {  
 *// out.write(buf, 0, len); 将数据写入到文件中.  
 // String 类的构造方法 : String(byte[] bytes, int offset, int length)* String str = new String(buf, 0, len);  
 System.*out*.println("str = " + str);  
 String ip = socket.getInetAddress().getHostAddress();  
 System.*out*.println("ip = " + ip);  
 }  
  
 *// 4. 后写 (获取socket通道的输出流)* OutputStream out = socket.getOutputStream();  
 out.write("不约, 叔叔.".getBytes());  
 out.flush();  
  
 *// 4.2 向socket通道中写出一个结束符* socket.shutdownOutput();  
  
 *// 5. 关闭服务端套接字* serverSocket.close();  
 }  
}

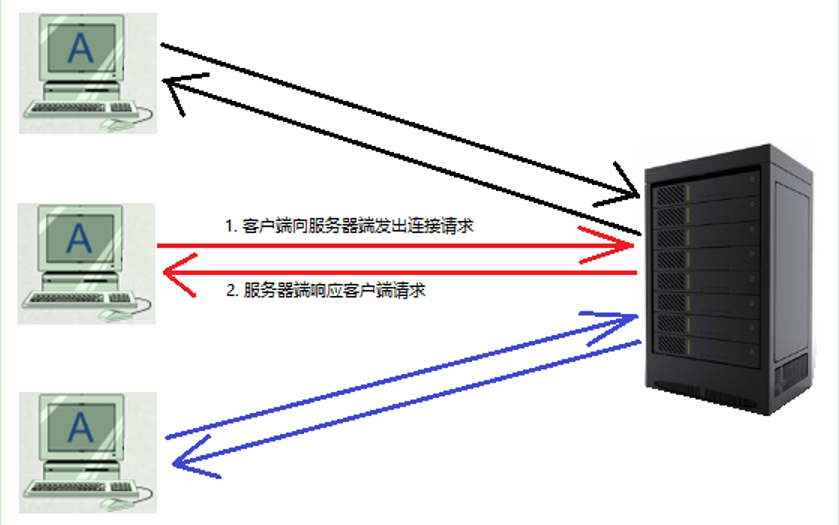
**客户端实现 :**

public class TestClient2 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个客户端套接字对象  
 // new Socket("localhost", 8888);  
 // new Socket("127.0.0.1", 8888);* Socket socket = new Socket("192.168.26.70", 8888); *// 尝试三次连接  
  
 // 2. 先写, 写入数据到通道中* OutputStream out = socket.getOutputStream();  
 out.write("小姐姐, 约吗 ?".getBytes());  
 out.flush();  
  
 *// 2.2 需要向通道中写出一个 `结束符`.* socket.shutdownOutput();  
  
  
  
 *// 3. 后读, 从通道中读取数据* InputStream in = socket.getInputStream();  
 byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = in.read(buf)) != -1) {  
 String str = new String(buf, 0, len);  
 System.*out*.println(str);  
 }  
  
 *// 4. 关闭客户端套接字* socket.close();  
 }  
}

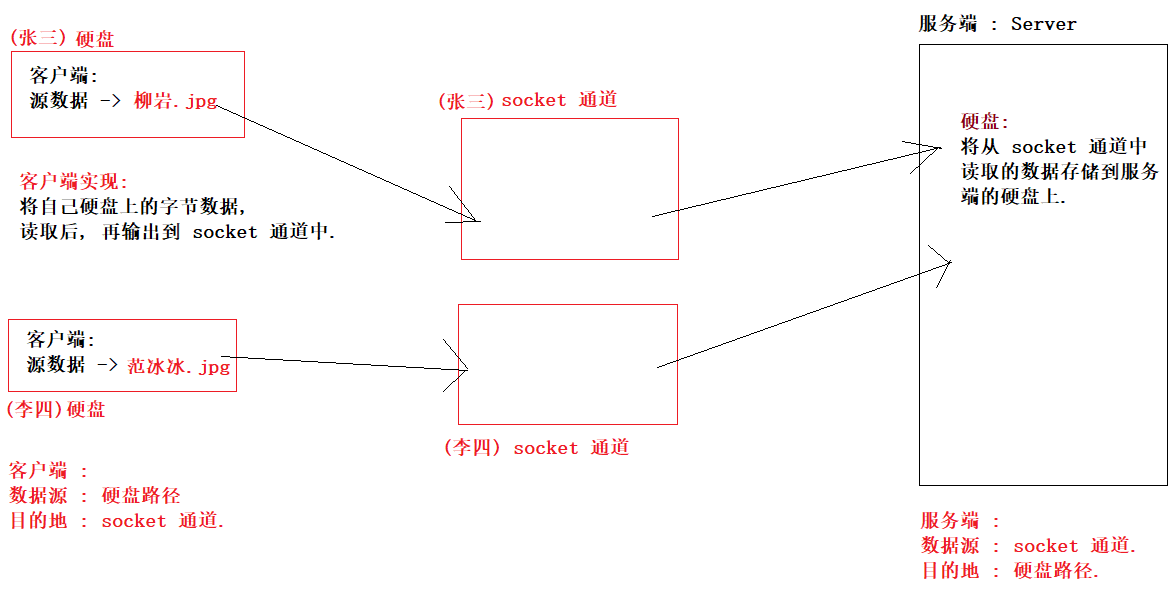




## 5. 综合案例\_文件上传 (重点)



**特别说明 :** 网络编程中, 文件数据一定要记得 `**刷新与关闭**`, 才能保证数据的及时与准确性.



实现方案一 :

问题 :

1. 图片名称问题.
2. 图片应该存储到服务端的指定文件夹中.
3. 服务端程序不应该只接收一次客户端就终止. 应该实现循环接收.

**客户端 :**

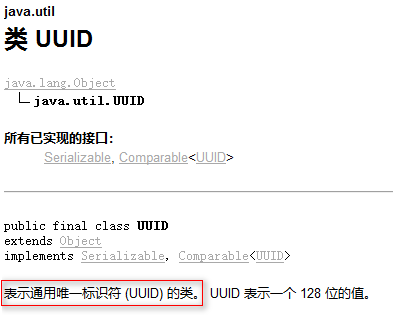
public class UploadClient1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个客户端套接字* Socket socket = new Socket("192.168.26.70", 8888);  
  
 *// 2. 创建两个高效的字节缓冲流 (BufferedInputStream / BufferedOutputStream)  
 // 2.1 创建一个高效的字节缓冲输入流, 与硬盘路径关联* String path = "C:\\Users\\xieco\\Desktop\\day08-File类、递归\\课程资料\\范冰冰.jpg";  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream(path));  
 *// 2.2 创建一个高效的字节缓冲输出流, 与 socket 通道实现关联* BufferedOutputStream bos = new BufferedOutputStream(socket.getOutputStream());  
  
 *// 2.3 实现读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 2.4 向通道中写出一个输出结束符* socket.shutdownOutput();  
  
 *// 3. 读取服务端返回的回送数据 (通道的输入流)* InputStream in = socket.getInputStream();  
 byte[] readBuf = new byte[1024];  
 int readLen = -1;  
 while ((readLen = in.read(readBuf)) != -1) {  
 String str = new String(readBuf, 0, readLen);  
 System.*out*.println("服务端回送数据 : " + str);  
 }  
  
 *// 4. 关闭客户端对象* bis.close();  
 socket.close();  
 }  
}

**服务端 :**

public class UploadServer1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待客户端连接* Socket socket = serverSocket.accept();  
  
 *// 3.1 创建一个高效的字节缓冲输入流, 与 socket 通道绑定.* BufferedInputStream bis = new BufferedInputStream(socket.getInputStream());  
 *// 3.2 创建一个高效的字节缓冲输出流, 与硬盘路径绑定.* BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("C:\\Users\\xieco\\Desktop\\柳岩.jpg"));  
  
 *// 3.3 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 4. 给客户端回送数据* OutputStream out = socket.getOutputStream();  
 out.write("上传成功!".getBytes());  
 *// 4.2 传输输出结束符* socket.shutdownOutput();  
  
 *// 5. 关闭服务端套接字* bos.close();  
 serverSocket.close();  
 }  
}

实现方案二 : 父目录文件夹与文件名称解决方案.

**UUID 测试类 :**

****



**服务端 :**

public class UploadServer1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待客户端连接* Socket socket = serverSocket.accept();  
  
 *// 3.1 创建一个高效的字节缓冲输入流, 与 socket 通道绑定.* BufferedInputStream bis = new BufferedInputStream(socket.getInputStream());  
  
 *// 步骤一 : 父目录文件夹 :* File parentFile = new File("C:\\Users\\xieco\\Desktop\\upload");  
 *// 步骤二 : 判断父目录是否存在* if (parentFile.exists() == false) {  
 *// 步骤三 : 不存在, 需要创建父目录* parentFile.mkdirs();  
 }  
 *// 3.2 创建一个高效的字节缓冲输出流, 与硬盘路径绑定.  
 // 方式一 : ip 地址  
 // String ip = socket.getInetAddress().getHostAddress();  
 // 方式二 : UUID 名称* String name = UUID.*randomUUID*().toString().replaceAll("-", "");  
  
 *// 步骤四 : 让文件名称与父目录实现关联, 从而创建一个 file 对象* File file = new File(parentFile, name + ".jpg");  
  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file));  
  
 *// 3.3 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 4. 给客户端回送数据* OutputStream out = socket.getOutputStream();  
 out.write("上传成功!".getBytes());  
 *// 4.2 传输输出结束符* socket.shutdownOutput();  
  
 *// 5. 关闭服务端套接字* bos.close();  
 serverSocket.close();  
 }  
}

实现方案三 : 解决服务端循环接收客户端连接. (必须使用多线程)

**服务端 :**

public class UploadServer1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 while (true) {  
 *// 2. 等待客户端连接* Socket socket = serverSocket.accept(); *// 循环接收客户端连接在 `主线程` 执行.  
  
 // 3. 创建一个图片上传任务类对象* UploadImageTask uploadImageTask = new UploadImageTask(socket);  
  
 *// 4. 将任务类对象与线程类对象实现关联* Thread uploadImageThread = new Thread(uploadImageTask, "图片上传线程");  
  
 *// 5. 启动子线程* uploadImageThread.start();  
 }  
  
 *// 服务端可以不关闭.  
 // serverSocket.close();* }  
}

**方式一 : 实现类 UploadPictureTask**

public class UploadImageTask implements Runnable {  
 *// 属性* private Socket socket;  
  
 *// 构造方法* public UploadImageTask(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 try {  
 *// 3.1 创建一个高效的字节缓冲输入流, 与 socket 通道绑定.* BufferedInputStream bis = new BufferedInputStream(socket.getInputStream());  
  
 *// 步骤一 : 父目录文件夹 :* File parentFile = new File("C:\\Users\\xieco\\Desktop\\upload");  
 *// 步骤二 : 判断父目录是否存在* if (parentFile.exists() == false) {  
 *// 步骤三 : 不存在, 需要创建父目录* parentFile.mkdirs();  
 }  
 *// 3.2 创建一个高效的字节缓冲输出流, 与硬盘路径绑定.  
 // 方式一 : ip 地址  
 // String ip = socket.getInetAddress().getHostAddress();* String name = UUID.*randomUUID*().toString().replaceAll("-", "");  
  
 *// 步骤四 : 让文件名称与父目录实现关联, 从而创建一个 file 对象* File file = new File(parentFile, name + ".jpg");  
  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file));  
  
 *// 3.3 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 *// 张三 -> 图片 (2秒) 读写是一个耗时操作. 耗时操作应该在 `子线程` 执行.* while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 4. 给客户端回送数据* OutputStream out = socket.getOutputStream();  
 out.write("上传成功!".getBytes());  
 *// 4.2 传输输出结束符* socket.shutdownOutput();  
  
 *// 5. 关闭服务端套接字* bos.close();  
 } catch (IOException e) {  
 System.*out*.println("上传失败!");  
 }  
 }  
}

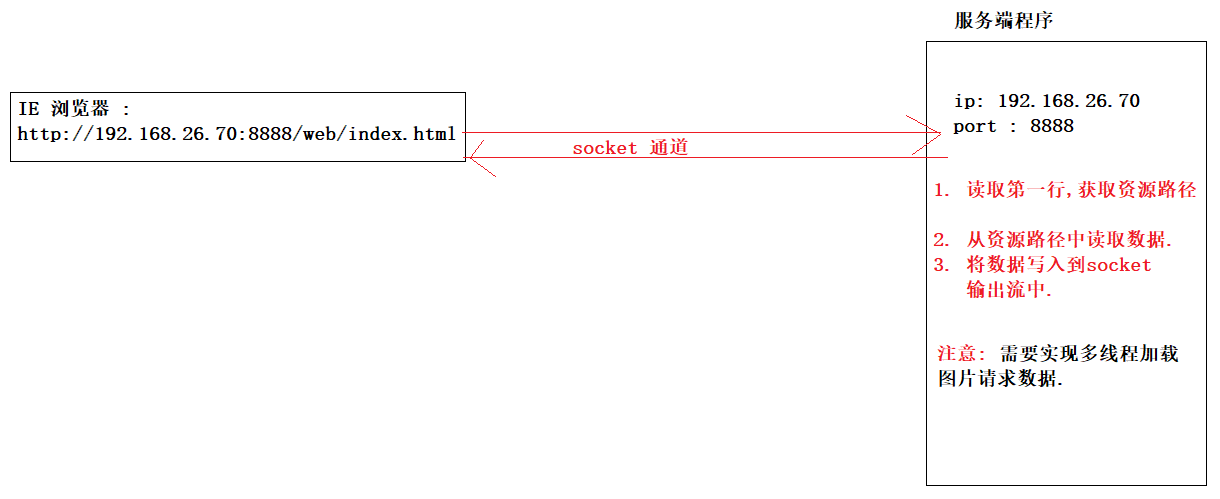
**方式二 : 匿名实现类 (不推荐)**

public class UploadServer1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 while (true) {  
 *// 2. 等待客户端连接* Socket socket = serverSocket.accept(); *// 循环接收客户端连接在 `主线程` 执行.  
  
 // 3. 创建一个图片上传任务类对象  
 // UploadImageTask uploadImageTask = new UploadImageTask(socket);  
  
 // 4. 将任务类对象与线程类对象实现关联* Thread uploadImageThread = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 *// 3.1 创建一个高效的字节缓冲输入流, 与 socket 通道绑定.* BufferedInputStream bis = new BufferedInputStream(socket.getInputStream());  
  
 *// 步骤一 : 父目录文件夹 :* File parentFile = new File("C:\\Users\\xieco\\Desktop\\upload");  
 *// 步骤二 : 判断父目录是否存在* if (parentFile.exists() == false) {  
 *// 步骤三 : 不存在, 需要创建父目录* parentFile.mkdirs();  
 }  
 *// 3.2 创建一个高效的字节缓冲输出流, 与硬盘路径绑定.  
 // 方式一 : ip 地址  
 // String ip = socket.getInetAddress().getHostAddress();* String name = UUID.*randomUUID*().toString().replaceAll("-", "");  
  
 *// 步骤四 : 让文件名称与父目录实现关联, 从而创建一个 file 对象* File file = new File(parentFile, name + ".jpg");  
  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file));  
  
 *// 3.3 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 *// 张三 -> 图片 (2秒) 读写是一个耗时操作. 耗时操作应该在 `子线程` 执行.* while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 4. 给客户端回送数据* OutputStream out = socket.getOutputStream();  
 out.write("上传成功!".getBytes());  
 *// 4.2 传输输出结束符* socket.shutdownOutput();  
  
 *// 5. 关闭服务端套接字* bos.close();  
 } catch (IOException e) {  
 System.*out*.println("上传失败!");  
 }  
 }  
 }, "图片上传线程");  
  
 *// 5. 启动子线程* uploadImageThread.start();  
 }  
  
 *// 服务端可以不关闭.  
 // serverSocket.close();* }  
}

**方式三 : Lambda 表达式 (不推荐)**

public class UploadServer1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 while (true) {  
 *// 2. 等待客户端连接* Socket socket = serverSocket.accept(); *// 循环接收客户端连接在 `主线程` 执行.  
  
 // 3. 创建一个图片上传任务类对象  
 // UploadImageTask uploadImageTask = new UploadImageTask(socket);  
  
 // 4. 将任务类对象与线程类对象实现关联* Thread uploadImageThread = new Thread(() -> {  
 try {  
 *// 3.1 创建一个高效的字节缓冲输入流, 与 socket 通道绑定.* BufferedInputStream bis = new BufferedInputStream(socket.getInputStream());  
  
 *// 步骤一 : 父目录文件夹 :* File parentFile = new File("C:\\Users\\xieco\\Desktop\\upload");  
 *// 步骤二 : 判断父目录是否存在* if (parentFile.exists() == false) {  
 *// 步骤三 : 不存在, 需要创建父目录* parentFile.mkdirs();  
 }  
 *// 3.2 创建一个高效的字节缓冲输出流, 与硬盘路径绑定.  
 // 方式一 : ip 地址  
 // String ip = socket.getInetAddress().getHostAddress();* String name = UUID.*randomUUID*().toString().replaceAll("-", "");  
  
 *// 步骤四 : 让文件名称与父目录实现关联, 从而创建一个 file 对象* File file = new File(parentFile, name + ".jpg");  
  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file));  
  
 *// 3.3 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 *// 张三 -> 图片 (2秒) 读写是一个耗时操作. 耗时操作应该在 `子线程` 执行.* while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 4. 给客户端回送数据* OutputStream out = socket.getOutputStream();  
 out.write("上传成功!".getBytes());  
 *// 4.2 传输输出结束符* socket.shutdownOutput();  
  
 *// 5. 关闭服务端套接字* bos.close();  
 } catch (IOException e) {  
 System.*out*.println("上传失败!");  
 }  
 }, "图片上传线程");  
  
 *// 5. 启动子线程* uploadImageThread.start();  
 }  
  
 *// 服务端可以不关闭.  
 // serverSocket.close();* }  
}

## 6. 模拟服务器 : (重点)



**浏览器访问服务器.**

**实现一 :** 完成浏览器访问自己编写的服务器, 查看浏览器附带的信息.

public class ServerTest1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 浏览器传输给服务器的数据格式都是 HTTP 协议固定的.  
  
 // 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待连接请求* Socket socket = serverSocket.accept();  
  
 *// 3. 接收数据* InputStream in = socket.getInputStream();  
 byte[] buf = new byte[1024];  
 int len = in.read(buf);  
 String str = new String(buf, 0, len);  
 System.*out*.println(str);  
  
 *// 4. 关闭服务端* serverSocket.close();  
 }  
}

**数据内容 :**

GET / HTTP/1.1 请求方式 资源路径 协议版本

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/17.17134

Accept-Language: zh-Hans-CN,zh-Hans;q=0.5

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Upgrade-Insecure-Requests: 1

Accept-Encoding: gzip, deflate

Host: localhost:8888

Connection: Keep-Alive

GET /web/index.html HTTP/1.1

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/17.17134

Accept-Language: zh-Hans-CN,zh-Hans;q=0.5

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Upgrade-Insecure-Requests: 1

Accept-Encoding: gzip, deflate

Host: localhost:8888

Connection: Keep-Alive

**实现二 :** 截取访问的资源路径.

**web/index.html 浏览器访问的资源路径.**

public class ServerTest1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 浏览器传输给服务器的数据格式都是 HTTP 协议固定的.  
  
 // 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待连接请求* Socket socket = serverSocket.accept();  
  
 *// 3. 接收数据* InputStream in = socket.getInputStream();  
  
 *// BufferedReader 类 readLine(); 方法* BufferedReader reader = new BufferedReader(new InputStreamReader(in));  
 String line = reader.readLine();  
 *// GET /web/index.html HTTP/1.1* String[] split = line.split(" ");  
 String path = split[1].substring(1);  
 System.*out*.println(path);  
  
 *// 4. 关闭服务端* serverSocket.close();  
 }  
}

**实现三 :** 服务器向浏览器写入访问资源数据.

// 写入HTTP协议响应头,固定写法

bos.write("HTTP/1.1 200 OK\r\n".getBytes());

bos.write("Content-Type:text/html\r\n".getBytes());

// 必须要写入空行,否则浏览器不解析

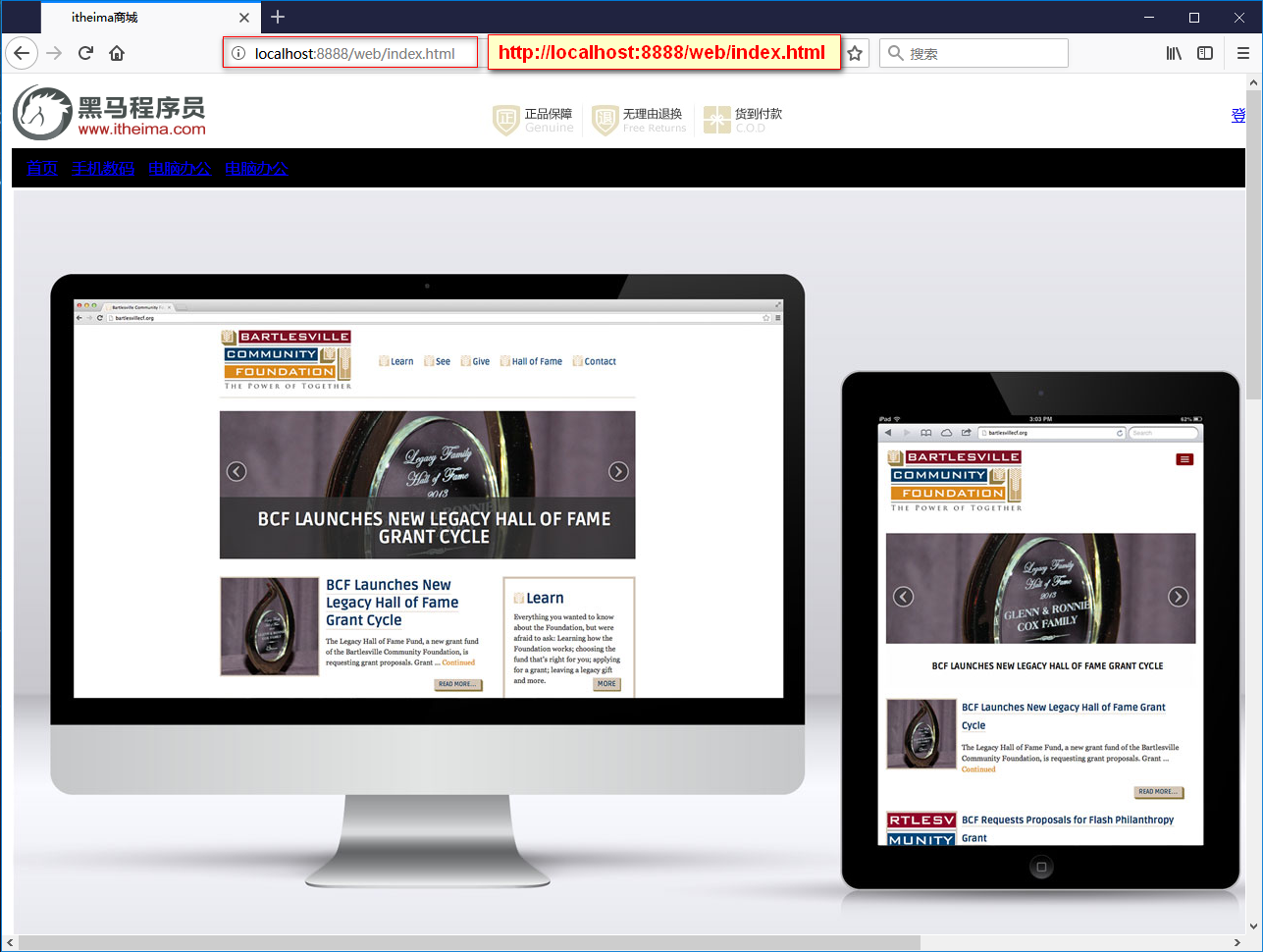
bos.write("\r\n".getBytes());

public class ServerTest1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 浏览器传输给服务器的数据格式都是 HTTP 协议固定的.  
  
 // 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 *// 2. 等待连接请求* Socket socket = serverSocket.accept();  
  
 *// 3. 接收数据* InputStream in = socket.getInputStream();  
  
 *// BufferedReader 类 readLine(); 方法* BufferedReader reader = new BufferedReader(new InputStreamReader(in));  
 String line = reader.readLine();  
 *// GET /web/index.html HTTP/1.1* String[] split = line.split(" ");  
 String path = split[1].substring(1); *// path  
  
 // 4. 关联两个高效的缓冲字节流对象* BufferedInputStream bis = new BufferedInputStream(new FileInputStream(path));  
 BufferedOutputStream bos = new BufferedOutputStream(socket.getOutputStream());  
  
 *// 先写出 HTTP 响应的格式 :  
 // 写入HTTP协议响应头,固定写法* bos.write("HTTP/1.1 200 OK\r\n".getBytes());  
 bos.write("Content-Type:text/html\r\n".getBytes());  
 *// 必须要写入空行,否则浏览器不解析* bos.write("\r\n".getBytes());  
  
 *// 5. 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 5.2 输出通道结束符* socket.shutdownOutput();  
  
 *// 6. 关闭服务端* serverSocket.close();  
 }  
}

**实现四 :** 多线程实现图片数据加载. (**静态内部类**)

浏览器访问服务器, 在图片中, 服务器必须要提供子线程实现图片加载. 另外 accept 必须循环接收, 因为每一张图片的加载, 浏览器就会发送一个新的请求.

public class ServerTest1 {  
 public static void main(String[] args) throws IOException {  
  
 *// 浏览器传输给服务器的数据格式都是 HTTP 协议固定的.  
  
 // 1. 创建一个服务端套接字* ServerSocket serverSocket = new ServerSocket(8888);  
  
 while (true) {  
  
 *// 2. 等待连接请求* Socket socket = serverSocket.accept();  
  
 *// 3. 创建一个任务类对象* WebImageTask webImageTask = new WebImageTask(socket);  
  
 *// 4. 将任务类对象与线程类对象实现关联* Thread webImageThread = new Thread(webImageTask, "网页图片线程");  
  
 *// 5. 启动线程对象* webImageThread.start();  
 }  
  
 *// 6. 关闭服务端  
 // serverSocket.close();* }  
  
 *// 定义一个静态内部类 :* private static class WebImageTask implements Runnable {  
  
 *// 属性* private Socket socket;  
  
 *// 构造方法* public WebImageTask(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 try {  
 *// 3. 接收数据* InputStream in = socket.getInputStream();  
  
 *// BufferedReader 类 readLine(); 方法* BufferedReader reader = new BufferedReader(new InputStreamReader(in));  
 String line = reader.readLine();  
 *// GET /web/index.html HTTP/1.1* String[] split = line.split(" ");  
 String path = split[1].substring(1); *// path  
  
 // 4. 关联两个高效的缓冲字节流对象* BufferedInputStream bis = new BufferedInputStream(new FileInputStream(path));  
 BufferedOutputStream bos = new BufferedOutputStream(socket.getOutputStream());  
  
 *// 先写出 HTTP 响应的格式 :  
 // 写入HTTP协议响应头,固定写法* bos.write("HTTP/1.1 200 OK\r\n".getBytes());  
 bos.write("Content-Type:text/html\r\n".getBytes());  
 *// 必须要写入空行,否则浏览器不解析* bos.write("\r\n".getBytes());  
  
 *// 5. 读写操作* byte[] buf = new byte[1024];  
 int len = -1;  
 while ((len = bis.read(buf)) != -1) {  
 bos.write(buf, 0, len);  
 bos.flush();  
 }  
  
 *// 5.2 输出通道结束符* socket.shutdownOutput();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}



说明 :

能够说出TCP协议下两个常用类名称.

1. 服务端套接字 ServerSocket

2. 客户端套接字 Socket

能够编写TCP协议下字符串数据传输程序. -> 对应 TCP 通信图解与简单的TCP程序编写案例.

能够理解TCP协议下文件上传案例.

说明1 : 客户端从自己的硬盘路径上读取资源数据. 然后写入到 socket 通道中, 传送给服务端接口.

说明2 : 服务端从 socket 通道中读取客户端传送的数据, 然后写入到服务端的硬盘路径中存储.

能够理解TCP协议下BS案例.

说明 : 使用浏览器, 输入访问格式 : <http://localhost:8888/web/index.html> 将请求发送为服务器程序, 然后服务器实现资源路径截取, 再通过IO流和socket 通道将数据回送给浏览器实现显示.

注意 : 图片资源都会重新发送请求实现数据加载, 因此, 服务端程序必须使用多线程, 完成资源传送, 并保持不断接收客户端发送的连接请求.

# 2. 函数式接口与方法引用 :

**面向对象 :**

1. 数据封装到对象中存储, 传输的时候, 可以直接传递对象, 因为数据在对象. (对象.属性)
2. 通过对象的对象去寻找对应的方法. substring 截取字符串.(String对象) ip地址 (InetAddress对象)

**() -> {}**  函数式编程的思想. (JDK 8 之后的新语法)

解决 : 匿名实现类书写的繁琐问题.

## 1. @FunctionalInterface 注解 :

注释 : 是程序猿与程序媛之间的交流工作. 主要是给人看的, 不是给编译器看的.

注解 : 是给编译器看.

@Override 重写 / 复写 (被override修饰的方法必须是父类中定义的方法,或者是接口中定义的方法)

@FunctionalInterface 函数式接口. (该接口有,且仅有一个抽象方法)

Lambda 表达式的使用前的两个前提条件 :

1. 必须有 `函数式接口`. (Supplier, Consumer, Function, Predicate)
2. 必须有方法使用 `函数式接口` 作为方法的参数. Stream流 (集合和数组)

满足前两个条件, 就可以在调用方法是, 传递 Lambda 表达式. (重点)

**函数式接口定义 :**

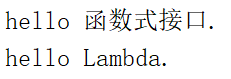
// 请问 : 该接口是函数式接口吗 ??? Lambda 表达式只能传递 `函数式` 接口  
// 注解 : 让编译器在编译阶段完成特定的语法检查  
@FunctionalInterface  
public interface MyFunctionalInterface {  
  
 void method(); // 默认修饰符 : public abstract  
}

**方法定义与传递 Lambda 表达式 :**

*// 模拟 : 使用函数式接口作为方法的参数  
// 理解 : 只要有接口类型作为方法的参数, 底层都会被自动调用.*public static void testMethod(boolean flag, MyFunctionalInterface inter) {  
 if (flag == true) {  
 inter.method(); *// 调用接口中抽象方法时, 执行的是实现类的方法.* }  
}

**演示效果 :**

public static void main(String[] args) {  
  
 // 重点 : 调用方法, 传递 Lambda 表达式  
 // 接口作为参数 : 传递该接口的实现类对象. (匿名实现类)  
 testMethod(true, new MyFunctionalInterface() {  
 // 实现了接口中的抽象方法.  
 // 请问 : 方法是在什么时候被执行的 ??? 被调用的时候.  
 @Override  
 public void method() {  
 System.out.println("hello 函数式接口.");  
 }  
 });  
  
 // Lambda 表达式 : 就是匿名实现类的简化写法.  
 /\*  
 1. () 抽象方法的参数列表  
 2. -> 分隔符  
 3. {} 抽象方法的方法实现体  
 \*/  
 testMethod(true, () -> { System.out.println("hello Lambda.");});  
}



## 2. 练习\_自定义函数式接口(无参无返回值)

**函数式接口定义 :**

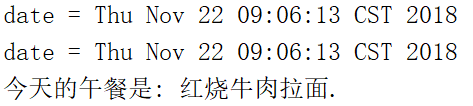
@FunctionalInterface  
public interface Eatable {  
  
 void eat();  
}

**方法定义与传递 Lambda 表达式 :**

*// 方法 : 使用函数式接口作为方法的参数列表*public static void testMethod(boolean hungry, Eatable eatable) {  
 *// eatable 接收的是 Lambda 表达式.* Date date = new Date();  
 System.*out*.println("date = " + date);  
  
 *// 判断* if (hungry) {  
 eatable.eat(); *// 执行 Lambda 表达式, 其实执行的就是 Lambda 表达式的方法体.* }  
}

**演示效果 :**

public static void main(String[] args) {  
  
 *// 重点 : 传递 Lambda 表达式  
 testMethod*(false, () -> {System.*out*.println("早餐吃包子."); });  
  
 *testMethod*(true, () -> { System.*out*.println("今天的午餐是: 红烧牛肉拉面."); });  
}



## 3. 练习\_自定义函数式接口(有参有返回值)

**函数式接口定义 :**

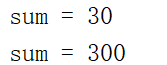
@FunctionalInterface  
public interface Calculator {  
  
 int calcSum(int num1, int num2);  
}

**方法定义与传递 Lambda 表达式 :**

*// 方法 : 为了使用 Lambda 表达式*public static void testMethod(int n1, int n2, Calculator calculator) {  
 int sum = calculator.calcSum(n1, n2);  
 System.*out*.println("sum = " + sum);  
}

**演示效果 :**

public static void main(String[] args) {  
  
 *// 调用方法 : 传递 Lambda 表达式  
 testMethod*(10, 20, (n1, n2) -> { return n1 + n2; });  
  
 *testMethod*(100, 200, (a, b) -> a + b);  
}



## 4. Lambda的延迟执行 :

**延迟执行的含义 :** 调用时才会被执行, 不调用不执行.

**普通传递方式说明 :**

*// 普通方法 :*public static void log(int level, String message) {  
 *// 判断* if (level == 1) {  
 System.*out*.println(message); *// 条件不成立, 不需要参数实现拼接的.* }  
}

**Lambda 表达式说明 :** **MessageBuilder**

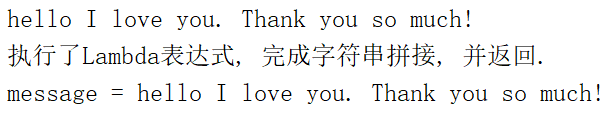
@FunctionalInterface  
public interface MessageBuilder {  
  
 String get();  
}

*// 函数式接口方法 :*public static void log(int level, MessageBuilder messageBuilder) {  
 *// 判断* if (level == 1) {  
 String message = messageBuilder.get(); *// 调用函数式接口的抽象方法, 其实就是执行 Lambda 方法体.* System.*out*.println(message);  
 }  
}

**Lambda传递方式说明 :**

public static void main(String[] args) {  
  
 *// 调用方法 :* String s1 = "hello ";  
 String s2 = "I love you.";  
 String s3 = "Thank you so much!";  
  
 *// 请问 : s1 + s2 + s3 在内存中是否已经开辟空间, 实现数据拼接了呢 ??? 已经实现了.  
 log*(1, s1 + s2 + s3); *// 提前执行了拼接操作.  
  
 // log(0, () -> { return s1 + s2 + s3; }); // 延迟执行了拼接操作  
  
 log*(1, () -> {  
 System.*out*.println("执行Lambda表达式, 完成字符串拼接并返回.");  
 return s1 + s2 + s3;  
 });  
}

**演示效果 :**



## 5. 使用Lambda作为参数和返回值 :

### 1 Runnable作为方法参数 :

**函数式接口定义 :**



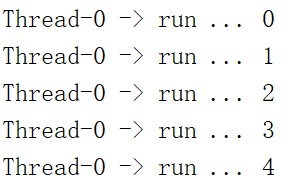
**方法定义:**



**传递 Lambda 表达式 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// Thread(Runnable target); 函数式接口  
 // 说明 : 使用函数式接口作为方法的参数类型. (传递Lambda表达式)* new Thread(() -> {  
 for (int i = 0; i < 5; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " -> run ... " + i);  
 }  
 }).start();  
 }  
}

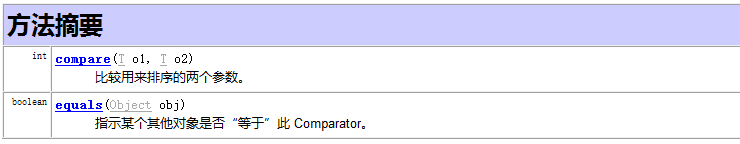
**演示效果 :**



### 2 Comparator作为方法返回值 :

String[] strArr = new String[]{ "Lily", "Ann", "Michal", "Jay", "Jerry", "JK", "Angel baby" };

**函数式接口定义 :**



**方法定义:**

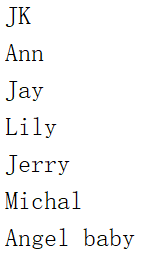




**传递 Lambda 表达式**

public class Test2 {  
 public static void main(String[] args) {  
  
 String[] strArr = new String[]{ "Lily", "Ann", "Michal", "Jay", "Jerry", "JK", "Angel baby" };  
  
 *// 调用 Arrays 数据工具类的 sort(T[] a, Comparator<? super T> c) 方法, 传递 Lambda 表达式  
 // Comparator 就是一个 `函数式` 接口, 因此可以直接传递 Lambda 表达式  
 // 抽象方法 : int compare(T o1, T o2);  
  
 // 需求 : 根据字符串的长度实现元素的排序. (升序)  
 // Arrays.sort(strArr, (s1, s2) -> {return s1.length() - s2.length();});* Arrays.*sort*(strArr, *compareByLength*());  
  
 for (String s : strArr) {  
 System.*out*.println(s);  
 }  
 }  
  
 *// 需求 : 使用 `函数式接口` 类型作为方法的返回值.  
 // 说明 : 可以直接返回一个 Lambda 表达式.* public static Comparator<String> compareByLength() {  
 return (s1, s2) -> {return s1.length() - s2.length(); };  
 }  
}

**演示效果 :**



## 6. 练习\_Lambda表达式作为参数与返回值(MySupplier)

**函数式接口定义 :**

@FunctionalInterface  
public interface MySupplier<T> {  
  
 *// 抽象方法* T get();  
}

**方法参数定义:**

*// 方法1 : 将 MySupplier 函数式接口作为方法的参数类型使用*public static void testMethod1(MySupplier<String> supplier) {  
 String result = supplier.get();  
 System.*out*.println("result = " + result);  
}

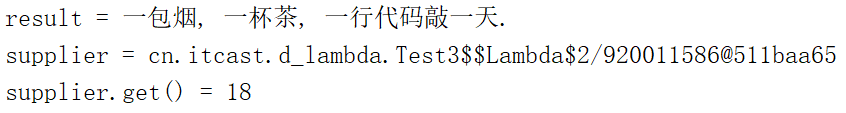
**返回值参数定义:**

*// 方法2 : 将 MySupplier 函数式接口作为方法的返回值使用*public static MySupplier<Integer> testMethod2(String key) {  
 HashMap<String, Integer> map = new HashMap<>();  
 map.put("迪丽热巴", 16);  
 map.put("古力娜扎", 18);  
 map.put("马尔扎哈", 100);  
  
 Integer value = map.get(key);  
 return () -> { return value; }; *// 返回的就类似于一个 `匿名实现类` 对象.*}

**传递 Lambda 表达式 :**

public static void main(String[] args) {  
  
 *// 调用方法一 : Lambda 作为参数  
 testMethod1*(() -> { return "一包烟, 一杯茶, 一行代码敲一天."; });  
  
 *// 调用方法二 : Lambda 作为返回值* MySupplier<Integer> supplier = *testMethod2*("古力娜扎");  
 System.*out*.println("supplier = " + supplier);  
 System.*out*.println("supplier.get() = " + supplier.get()); *// 调用接口的方法, 就执行了Lambda表达式的方法体.*}

**演示效果 :**



## 7. 方法引用 (冗余的Lambda场景) :

**格式 : 对象名::方法名**

**准备** : Printable 函数式接口中的 print 有参无返回值方法.

**函数式接口定义 :**

@FunctionalInterface  
public interface Printable<T> {  
  
 void print(T t);  
}

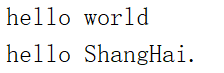
**方法定义:**

*// 方法 : Lambda 作为方法参数传递*public static void testMethod(String str, Printable<String> printable) {  
 printable.print(str);  
}

**传递 Lambda 表达式 :**

public static void main(String[] args) {  
  
 // 调用方法 :  
 testMethod("hello world.", (s) -> { System.out.println(s); });  
  
 // 对象方法 : 对象调用的方法. (非静态方法)  
 // 类方法 : 类名直接调用方法. (静态方法)  
  
 // Lambda 表达式的方法体实现与 PrintStream 类的对象方法 println 方法很相似.  
 // 说明 : 方法引用, 其实就是 Lambda 表达式的孪生兄弟.  
 // 书写格式 : 对象名::方法名  
  
 /\*  
 请问 : Printable 函数式接口的 void print(T t); 方法与 ps 对象的 println 方法有什么关系 ???  
  
 PrintStream 类的 void println(String x) 方法.  
  
 因为这两个方法的 `参数列表和返回值类型` 是一致的. 因为可以将 Lambda 表达式修改为 `方法引用`.  
 \*/  
 PrintStream ps = System.out;  
 testMethod("hello method reference.", ps::println);  
  
 testMethod("hello 方法引用.", System.out::println);  
}

**演示效果 :**



## 8. 通过对象名引用成员方法 :

**准备1 :** Printable 函数式接口中的 print 有参无返回值方法.

**准备2 :** ObjectMethodRef 类中的 printUpperCase 有参无返回值方法.

**函数式接口定义 :**

@FunctionalInterface  
public interface Printable<T> {  
  
 void print(T t);  
}

**ObjectMethodRef 类定义 :**

public class ObjectMethodRef {  
  
 *// 将参数转换为大写输出* public void printUpperCase(String str) {  
 String toUpperCase = str.toUpperCase();  
 System.*out*.println(toUpperCase);  
 }  
}

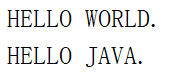
**方法定义:**

*// 方法 : 将 Printable 函数式接口作为参数列表*public static void testMethod(String str, Printable<String> printable) {  
 printable.print(str);  
}

**传递 Lambda 表达式:**

public static void main(String[] args) {  
  
 // 需求 : 调用方法, 传入一个字符串, 并使用 Lambda 表达式将字符串转换为大写输出.  
 /\*testMethod("hello world.", (s) -> {  
 String upperCase = s.toUpperCase();  
 System.out.println(upperCase);  
 });\*/  
  
 testMethod("hello world.", (s) -> { System.out.println(s.toUpperCase()); });  
  
 // 方法引用 :  
 /\*  
 思考1 : 能不能找到一个方法, 与 Lambda 表达式的方法体执行逻辑一样的方法??? printUpperCase方法  
 思考2 : 该方法的参数列表与返回值类型是否和 `函数式接口` 中的抽象方法一致 ???  
  
 Printable : void print(T t);  
 ObjectMethodRef : void printUpperCase(String str)  
  
 总结 : 如果前两条思考都能够满足, 就可以使用方法引用来实现传递.  
 \*/  
 // testMethod("hello world.", 对象名::对象方法);  
 testMethod("hello java.", new ObjectMethodRef()::printUpperCase);  
}

**演示效果 :**



## 9. 对象名引用成员方法(练习)

**准备1 :** WorkHelper 函数式接口中的 help 有参无返回值方法.

**准备2 :** Assistant 类中的 dealFile 有参无返回值方法.

**函数式接口定义 :**

@FunctionalInterface  
public interface WorkHelper {  
  
 void help(String name);  
}

**Assistant 类定义 :**

public class Assistant {  
  
 *// 行为 : dealFile 处理文件* public void dealFile(String filename) {  
 System.*out*.println("帮忙处理<" + filename + ">文件.");  
 }  
}

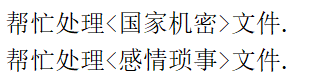
**方法定义:**

*// 方法 : 将函数式接口作为方法的参数*public static void testMethod(String name, WorkHelper workHelper) {  
 workHelper.help(name);  
}

**传递 Lambda 表达式:**

public class Test3 {  
 public static void main(String[] args) {  
  
 *// 调用, 传递 Lambda 表达式  
 testMethod*("国家机密", (s) -> { System.*out*.println("帮忙处理<" + s + ">文件."); });  
  
 *// 方法引用 :  
 /\*  
 条件1 : 必须已经存在一个方法, 与 Lambda 表达式的实现体是一致的. dealFile 方法.  
 条件2 : 即将被引用的方法与函数式接口中的抽象方法一致吗 ??? (参数列表和返回值类型)  
  
 WorkHelper : void help(String name);  
 Assistant : void dealFile(String filename)  
  
 总结 : 功能一致, 并且 方法签名 [返回值类型 方法名(参数列表)] 也一致, 就可以使用方法引用了.  
 格式 : 对象::对象方法名  
 \*/* Assistant assistant = new Assistant();  
 *testMethod*("感情琐事", assistant::dealFile);  
 }  
  
 *// 方法 : 将函数式接口作为方法的参数* public static void testMethod(String name, WorkHelper workHelper) {  
 workHelper.help(name);  
 }  
}

**演示效果 :**



## 10. 通过类名引用类方法 :

**准备 :** Calculator函数式接口中的 calc有参有返回值方法. (Math::abs 实现引用)

**函数式接口定义 :**

@FunctionalInterface  
public interface Calculator {  
  
 int calcAbsolute(int num);  
}

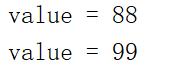
**方法定义:**

*// 方法 :*public static int testMethod(int number, Calculator calculator) {  
 int result = calculator.calcAbsolute(number);  
 return result;  
}

**传递 Lambda 表达式:**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 需求 : 传递第一个参数, 并获取第一个参数操作完成后的返回的绝对值  
  
 // 调用, 并传递 Lambda 表达式* int value = *testMethod*(-88, (n) -> {return Math.*abs*(n); });  
 System.*out*.println("value = " + value);  
  
 */\*  
 思考1 : 系统有没有提供一个方法与 Lambda 表达式的实现体是一致的 ??? 有. abs  
 思考2 : 系统提供的该方法与函数式接口中的抽象方法 `签名(返回值类型和参数列表)` 是一致的吗 ???  
  
 Calculator : int calcAbsolute(int num);  
 Math : int abs(int a)  
  
 总结 : 如果前两条条件满足, 就可以使用 `方法引用`.  
 \*/  
 // 格式 : 类名::类方法* int value2 = *testMethod*(-99, Math::*abs*);  
 System.*out*.println("value2 = " + value2);  
 }  
  
 *// 方法 :* public static int testMethod(int number, Calculator calculator) {  
 int result = calculator.calcAbsolute(number);  
 return result;  
 }  
}

**演示效果 :**



## 11. 类名引用静态方法(练习)

**格式 :** 类名::类方法

**准备1 :** StringChecker函数式接口的 checkString有参有返回值方法.

**准备2 :** StringUitls自定义字符串工具类的 isBlank有参有返回值方法. (Lambda传递与方法引用) `类`

**函数式接口定义 :**

@FunctionalInterface  
public interface StringChecker {  
  
 boolean checkString(String str);  
}

**StringUtils 类定义 :**

// 工具类一般都是提供 `静态方法`. 直接可以使用类名调用  
public class StringUtils {  
  
 // 判断字符串是否为空的方法  
 public static boolean isBlank(String str) {  
 // boolean result1 = str == null;  
 // boolean result2 = str.trim().equals("");  
  
 return str == null || str.trim().equals("");  
 }  
}

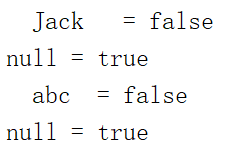
**方法定义:**

*// 方法 : 函数式接口作为方法的参数*public static void testMethod(String str, StringChecker stringChecker) {  
 boolean result = stringChecker.checkString(str);  
 System.*out*.println(str + " = " + result);  
}

**传递 Lambda 表达式:**

public class Test2 {  
 public static void main(String[] args) {  
  
 *// 需求 : 传入一个字符串, 使用 Lambda 表达式来判断该字符串是否为空. (null, "");  
  
 // 调用方法 :  
 testMethod*(" Jack ", (s) -> { return s == null || s.trim().equals(""); });  
 *testMethod*(null, (s) -> { return s == null || s.trim().equals(""); });  
  
 *// 方法引用 :  
 /\*  
 思考1 : 现成能不能找到一个方法, 与 Lambda 实现体执行逻辑一致的方法 ??? isBlank  
 思考2 : 即将被引用的方法, 与函数式接口中的抽象方法签名一致吗 ???  
  
 StringChecker : boolean checkString(String str);  
 StringUtils : boolean isBlank(String str)  
  
 总结 : 如果满足前两个条件, 就可以传递 `方法引用`.  
 \*/  
 // testMethod(" abc ", 类名::类方法名);  
 testMethod*(" abc ", StringUtils::*isBlank*);  
 *testMethod*(null, StringUtils::*isBlank*);  
 }  
  
 *// 方法 : 函数式接口作为方法的参数* public static void testMethod(String str, StringChecker stringChecker) {  
 boolean result = stringChecker.checkString(str);  
 System.*out*.println(str + " = " + result);  
 }  
}

**演示效果 :**



## 12. 通过super引用成员方法 :

**格式 : super::父类方法名**

**准备1 :** Greetable函数式接口中定义 greet 无参无返回值方法.

**准备2 :** 父类 Human 与子类 Man.

**子父类继承关系 :**

*// 人类*public class Human {  
  
 public void sayHi() {  
 System.*out*.println("大家好, 我是人类.");  
 }  
}

public class Man extends Human {  
}

**函数式接口定义 :**

@FunctionalInterface  
public interface Greetings {  
  
 void greet();  
}

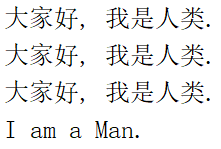
**Man类的方法定义:**

*// 方法 : 函数式接口作为方法的参数*private void testMethod(Greetings greetings) {  
 greetings.greet();  
}

**传递 Lambda 表达式 :**

public class Man extends Human {  
  
 *// 行为 : sayHello* public void sayHello() {  
  
 *// 调用 testMethod 方法, 传递 Lambda 表达式. 实现和父类 sayHi 方法一样的效果.  
 // 方式一 : Lambda 表达式* testMethod(() -> { System.*out*.println("大家好, 我是人类."); });  
  
 *// 方式二 : super.方法名();* testMethod(() -> { super.sayHi(); });  
  
 *// 方式三 : 方法引用 super::父类方法名* testMethod(super::sayHi);  
  
 System.*out*.println("I am a man.");  
 }  
  
 *// 方法 : 函数式接口作为方法的参数* private void testMethod(Greetings greetings) {  
 greetings.greet();  
 }  
}

**演示效果 :**



## 13. 通过this引用成员方法 :

**准备1 :** Richable函数式接口中定义 buy 无参无返回值方法.

**准备2 :** 类 Husband 提供 beHappy方法.

**函数式接口定义 :**

@FunctionalInterface  
public interface Richable {  
  
 void buy();  
}

**方法定义:**

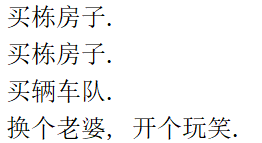
*// 结婚 : 买东西*public void merry(Richable richable) {  
 richable.buy();  
}

**传递 Lambda 表达式**

public class Husband {  
  
 public void beHappy() {  
  
 *// 要快乐, 就结婚  
 // 方式一 : 传递一个 Lambda 表达式* merry(() -> { System.*out*.println("买栋房子."); });  
  
 *// 方式二 : 方法引用 this::方法名* merry(this::buyHouse);  
  
 *// 方式三 : 买辆车对* merry(this::buyCar);  
  
 *// 方式四 : 终极行为* merry(this::changeWife); *// this对象是非静态的, 不可以引用静态方法.* }  
  
 *// 结婚 : 买东西* public void merry(Richable richable) {  
 richable.buy();  
 }  
  
 *// 买房子* private void buyHouse() {  
 System.*out*.println("买栋房子.");  
 }  
  
 private void buyCar() {  
 System.*out*.println("买辆车队.");  
 }  
  
 private void changeWife() {  
 System.*out*.println("换个老婆, 开个玩笑.");  
 }  
}

**演示效果 :**

public class Test1 {  
 public static void main(String[] args) {  
  
 Husband husband = new Husband();  
 husband.beHappy();  
 }  
}



## 14. 类的构造方法引用 :

**构造方法的作用 :** 创建当前类的对象.

**格式 :** 类名::new

**准备1 :** 类 Person, 属性 name.

**准备2 :** PersonBuilder函数式接口中定义 buildPerson有参有返回值方法.

**Person 类定义 :**

public class Person {  
 private String name;  
 private int age;  
  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
 public Person() {  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**函数式接口定义 :**

@FunctionalInterface  
public interface PersonBuilder {  
  
 Person buildPerson(String name, int age);  
}

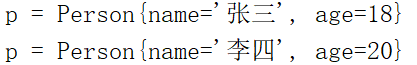
**方法定义:**

*// 方法 : 函数式接口作为方法的参数*public static void testMethod(String name, int age, PersonBuilder personBuilder) {  
 Person p = personBuilder.buildPerson(name, age);  
 System.*out*.println("p = " + p);  
}

**传递 Lambda 表达式:**

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 调用, 并传递参数  
 testMethod*("张三", 18, (name, age) -> { return new Person(name, age); });  
  
 *// 特殊语法 : 构造方法引用  
 // 格式 : 类名::new  
 // 请问 : Person::new 调用的是 Person 类的哪个构造方法 ???  
 // 回答 : 依赖于函数式接口中抽象方法的参数列表 ??? (String name, int age)  
 testMethod*("李四", 20, Person::new);  
 }  
  
 *// 方法 : 函数式接口作为方法的参数* public static void testMethod(String name, int age, PersonBuilder personBuilder) {  
 Person p = personBuilder.buildPerson(name, age);  
 System.*out*.println("p = " + p);  
 }  
}

**演示效果 :**



## 15. 数组的构造方法引用 :

**准备 :** ArrayBuilder函数式接口中定义 buildArray有参有返回值方法.

**函数式接口定义 :**

@FunctionalInterface  
public interface ArrayBuilder {  
  
 int[] buildArray(int length);  
}

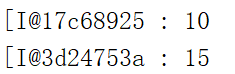
**方法定义:**

*// 方法: 函数式接口作为方法的参数类型*public static void testMethod(int len, ArrayBuilder arrayBuilder) {  
 int[] arr = arrayBuilder.buildArray(len);  
 System.*out*.println(arr + " : " + arr.length);  
}

**传递 Lambda 表达式:**

public class Test1 {  
 public static void main(String[] args) {  
  
 // 数组是一个特殊的引用类型. 标记: []  
 // 方式一 : Lmabda 表达式  
 testMethod(10, (len) -> { return new int[len]; });  
  
 // 方式二 : 方法引用 数组名::new int[], float[], String[], Student[]  
 testMethod(15, int[]::new);  
 }  
  
 // 方法: 函数式接口作为方法的参数类型  
 public static void testMethod(int len, ArrayBuilder arrayBuilder) {  
 int[] arr = arrayBuilder.buildArray(len);  
 System.out.println(arr + " : " + arr.length);  
 }  
}

**演示效果 :**



内容小结说明 :

Lambda 表达式 : () -> {} 入口, 查看是否编写正确. 参数列表和返回值需要与抽象方法保持一致.

方法引用 :

步骤一 : 考虑现成是否有一个方法与 Lambda 实现体执行的逻辑是一致.

步骤二 : 该方法是否与函数式接口中的抽象方法 `签名` 一致.

1. 能够使用@FunctionalInterface注解

含义 : 实现编译时期的语法检查, 该接口仅能存在一个抽象方法. 有, 且仅有一个.

1. 能够理解Lambda延迟执行的特点.

含义 : Lambda 表达式的实现体, 只有在被调用时, 才会执行. 如果内部条件不成立, 不调用时, Lambda 表达式是不会被执行. 这就是延迟执行的效果.

1. 能够使用Lambda作为方法的参数

含义 : 只有函数式接口作为方法的参数时, 才可以在调用方法时, 传递Lambda表达式. 格式: () -> {}

1. 能够使用输出语句的方法引用

含义 : System.out::println 输出语句的方法引用.

格式 : 对象名::对象方法 因为 System.out 返回的是 PrintStream 类型的对象. println 是一个对象方法. (非静态方法)

1. 能够通过4种方式使用方法引用

含义 : 方法引用建立在 Lambda 表达式的基础上.

格式:

类名::类方法

对象名::对象方法

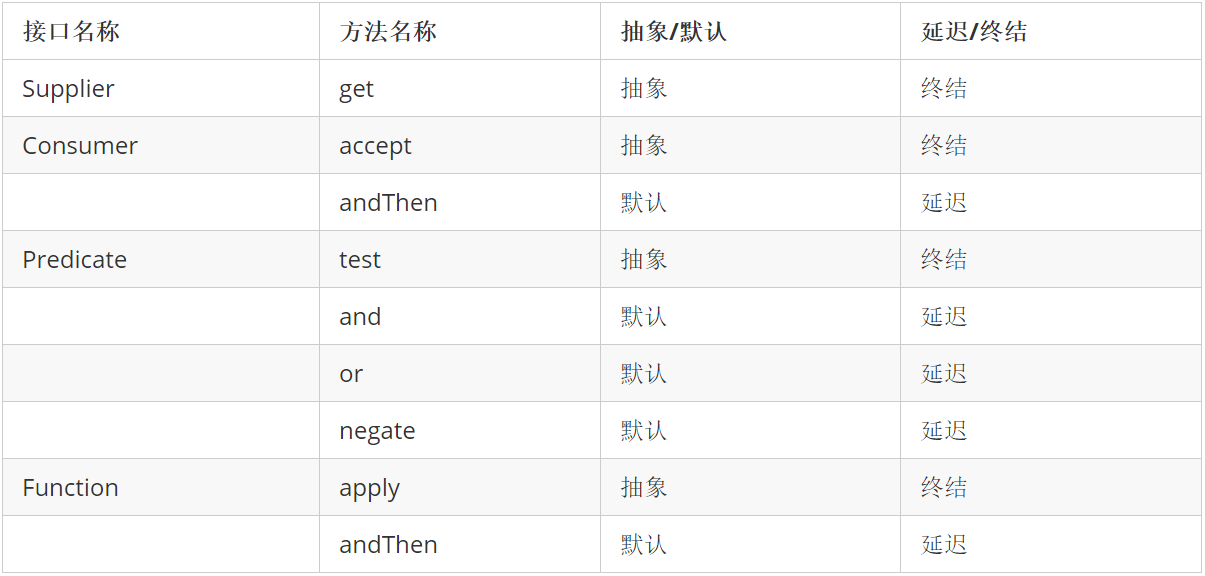
super::父类方法

this::本类方法

类名::new 构造方法引用

数组名::new 数组构造方法引用 (int[], float[], String[], Student[] …)

# 3. 常用函数式接口 (四大核心接口)



**系统提供的四大核心函数式接口 :**

记忆规律 :

Supplier<T> 供给型接口. 特点: 无中生有.

抽象方法 : T get();

Consumer<T> 消费型接口. 特点: 有去无回

抽象方法 : void accpet(T);

Predicate<T> 断言型接口. 特点: 元芳, 你怎么看? (给元芳一个数据,元芳告诉你成立不成立.)

抽象方法 : boolean test(T);

Function<T, R> 函数型接口. 特点: 有去有回.

抽象方法 : R apply(T);

## 1. Supplier 供给型接口 :

### 1. 抽象方法 : get

**定义方法 : 获取一个字符串**

*// 定义一个方法, 使用 Supplier 作为参数*public static void getString(Supplier<String> supplier) {  
 String s = supplier.get();  
 System.*out*.println(s);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 调用 :  
 getString*(() -> { return "hello world."; });  
}



### 2. 练习 : 求数组元素最大值

**定义方法 : 获取数组最大值**

*// 方法 :*public static void getMaxFromArray(Supplier<Integer> supplier) {  
 Integer value = supplier.get();  
 System.*out*.println(value);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 int[] arr = { 88, 66, 99, 33, 77, 55 };  
  
 *// 调用方法, 获取数组的最大值  
 getMaxFromArray*(() -> {  
 int max = arr[0];  
 for (int i = 1; i < arr.length; i++) {  
 if (max < arr[i]) {  
 max = arr[i];  
 }  
 }  
 return max;  
 });  
}



## 2. Consumer 消费型接口 :

### 1 抽象方法 : accept

**定义方法 : 消费一个字符串**

*// Consumer 消费型接口*public static void printString(String str, Consumer<String> consumer) {  
 consumer.accept(str);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 调用  
 printString*("Hello, 柳岩.", (s) -> { System.*out*.println("认识了一个新朋友." + s);});  
}



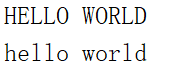
### 2 默认方法 : andThen

**定义方法 : 消费一个字符串, 先大写, 再小写**

public static void testString(String str, Consumer<String> one, Consumer<String> two) {  
 *// 含义 : 先让 one 对象 str 进行消费, 然后再让 two 对象 str 进行消费.* one.andThen(two).accept(str);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 需求 : 调用方法, 传递一个字符串, 先大写消费该字符串, 然后再小写消费该字符串  
 testString*("Hello World.",  
 (s) -> { System.*out*.println(s.toUpperCase());},  
 (s) -> { System.*out*.println(s.toLowerCase());});  
}



### 3 练习 : 格式化打印信息

String[] strArr = {"歌神,张学友", "舞王,郭富城", "综合,刘德华", "文艺,黎明"};

**定义方法 : 按照定义格式, 输出信息.**

昵称:歌神 = 张学友(真实姓名)

昵称:舞王 = 郭富城(真实姓名)

昵称:综合 = 刘德华(真实姓名)

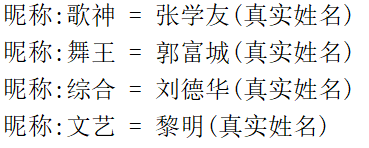
昵称:文艺 = 黎明(真实姓名)

**定义方法 :**

*// 方法 : Consumer 函数式接口作为参数类型*public static void testString(String[] strArr, Consumer<String> one, Consumer<String> two) {  
 *// 遍历字符串数组* for (String s : strArr) {  
 one.andThen(two).accept(s);  
 }  
}

**调用传递 :**

public static void main(String[] args) {  
  
 String[] strArr = {"歌神,张学友", "舞王,郭富城", "综合,刘德华", "文艺,黎明"};  
  
 /\*  
 昵称:歌神 = 张学友(真实姓名)  
 昵称:舞王 = 郭富城(真实姓名)  
 昵称:综合 = 刘德华(真实姓名)  
 昵称:文艺 = 黎明(真实姓名)  
 \*/  
 // "歌神,张学友" s.split(",")[0] s.split(",")[1]  
 testString(strArr,  
 (s) -> { System.out.print("昵称:" + s.split(",")[0] + " = ");},  
 (s) -> { System.out.println(s.split(",")[1] + "(真实姓名)");});  
  
}



## 3 Predicate断言型接口 :

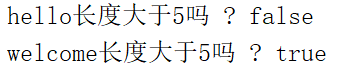
### 1 抽象方法 : test

**定义方法 : 判断字符串长度**

*// Predicate 断言型接口*public static void testStringLength(String str, Predicate<String> predicate) {  
 boolean result = predicate.test(str);  
 System.*out*.println(str + "长度大于5吗 ? " + result);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 调用 :  
 testStringLength*("hello", (s) -> { return s.length() > 5; });  
 *testStringLength*("welcome", (s) -> { return s.length() > 5; });  
}



### 2 默认方法 : and (并且) &&

**定义方法 : 判断字符串是否同时包含 H 和 W.**

public static void testString(String str, Predicate<String> one, Predicate<String> two) {  
 boolean result = one.and(two).test(str);  
 System.*out*.println(str + "是否同时包含大写的 `H 和 W`. " + result);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 需求 : 判断传入的字符串中是否同时包含大写的 `H 和 W`.  
 testString*("Hello world.",  
 (s) -> { return s.contains("H"); },  
 (s) -> { return s.contains("W"); });  
  
}



### 3 默认方法 : or (或者) ||

**定义方法 : 判断字符串是否包含 H 或者 W.**

public static void testString(String str, Predicate<String> one, Predicate<String> two) {  
 boolean result = one.or(two).test(str);  
 System.*out*.println(str + "是否包含大写的 `H` 或者大写的 W`. " + result);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 需求 : 是否包含大写的 `H` 或者大写的 W`.  
 testString*("Hello world.",  
 (s) -> { return s.contains("H"); },  
 (s) -> { return s.contains("W"); });  
  
}



### 4 默认方法 : negate (取反) !

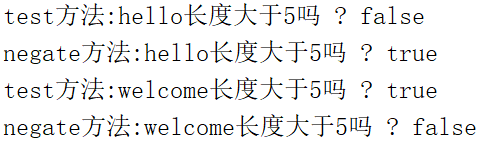
nagete 获取的结果刚好和 test 相反.

**定义方法 : 判断结果取反.**

*// Predicate 断言型接口*public static void testStringLength(String str, Predicate<String> predicate) {  
  
 boolean result = predicate.test(str);  
 System.*out*.println("test方法:" + str + "长度大于5吗 ? " + result);  
  
 result = predicate.negate().test(str); *// && and || or ! negate* System.*out*.println("negate方法:" + str + "长度大于5吗 ? " + result);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// 调用 :  
 testStringLength*("hello", (s) -> { return s.length() > 5; });  
 *testStringLength*("welcome", (s) -> { return s.length() > 5; });  
  
}



### 5 练习 : 集合信息筛选

需要同时满足两个条件：

1. 必须为女生；
2. 姓名为4个字。

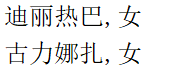
String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };

**定义方法 : 过滤数组元素, 并返回集合对象**

public static void testString(String[] strArr, Predicate<String> one, Predicate<String> two) {  
 for (String s : strArr) {  
 *// 判断* if (one.and(two).test(s)) {  
 System.*out*.println(s);  
 }  
 }  
}

**调用传递 :**

public static void main(String[] args) {  
  
 String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };  
  
 /\*  
 需要同时满足两个条件：  
 1. 必须为女生；  
 2. 姓名为4个字。  
 \*/  
 // 迪丽热巴,女 1. name.split(",")[1] -> "女" 2. name.split(",")[0] -> 迪丽热巴  
 testString(array,  
 (name) -> { return name.split(",")[1].equals("女"); },  
 (name) -> { return name.split(",")[0].length() == 4; });  
  
}



## 4 Function 函数型接口 :

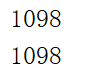
### 1 抽象方法 : apply

**定义方法 : 将字符串转换为整型返回**

*// 方法 :*public static void testString(String str, Function<String, Integer> function) {  
 Integer value = function.apply(str);  
 System.*out*.println(value + 100);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 // 需求 : 传递一个字符串, 将字符串转换为整型, 并运算.  
 testString("998", (str) -> { return Integer.parseInt(str); });  
  
 /\*  
 方法引用 :  
 Function : Integer apply(String t);  
 Integer : int parseInt(String s)  
 \*/  
 // testString("998", 类名::类方法);  
 testString("998", Integer::parseInt);  
}



### 2 默认方法 : andThen

**定义方法 : 先将字符串转换为整型, 然后再计算**

public static void testString(String str, Function<String, Integer> one, Function<Integer, Integer> two) {  
 Integer value = one.andThen(two).apply(str);  
 System.*out*.println("value = " + value);  
}

**调用传递 :**

public static void main(String[] args) {  
  
 *// "998" -> 998  
 // 998 + 2 -> 1000  
 testString*("998",  
 (str) -> { return Integer.*parseInt*(str); }, *// Integer::parseInt* (num) -> { return num + 2; });  
}



### 3 练习 : 自定义函数模型拼接

按照顺序执行多个操作：

1. 将字符串截取数字年龄部分，得到字符串；
2. 将上一步的字符串转换成为int类型的数字；
3. 将上一步的int数字累加10，得到结果int数字。

String str = "赵丽颖,18";

**定义方法 :**

public static void testString(String str, Function<String, String> one, Function<String, Integer> two,  
 Function<Integer, Integer> three) {  
 Integer result = one.andThen(two).andThen(three).apply(str);  
 System.*out*.println(result);  
}

**调用传递 :**

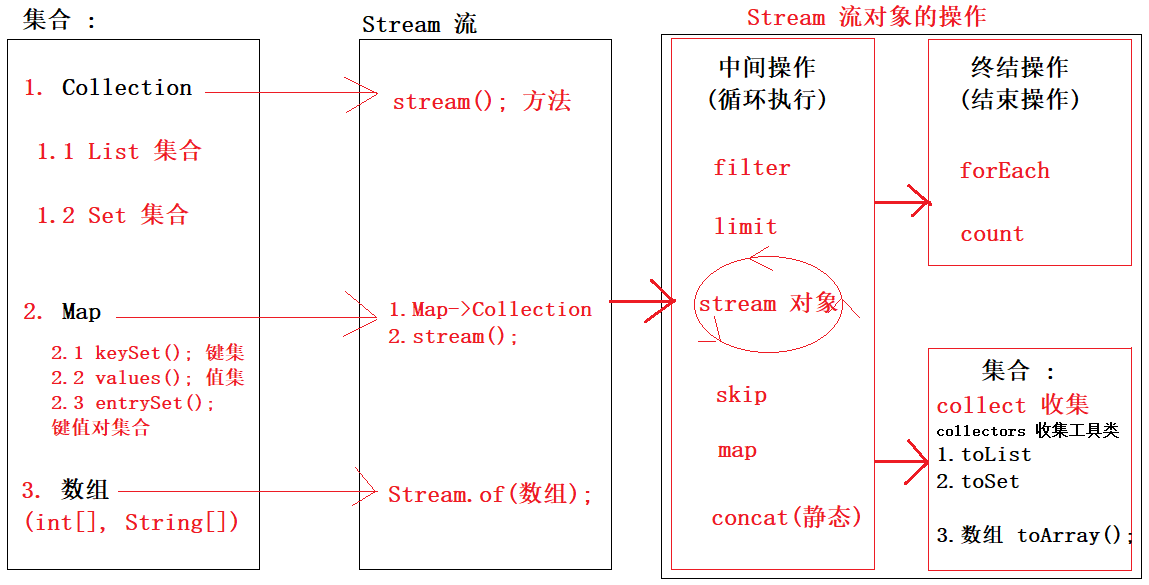
public static void main(String[] args) {  
  
 String str = "赵丽颖,18";  
  
 /\*  
 按照顺序执行多个操作：  
 1. 将字符串截取数字年龄部分，得到字符串； <String, String>  
 2. 将上一步的字符串转换成为int类型的数字； <String, Integer>  
 3. 将上一步的int数字累加10，得到结果int数字。 <Integer, Integer>  
 \*/  
 testString(str,  
 (s) -> { return s.split(",")[1]; }, // "18"  
 (s) -> { return Integer.parseInt(s); }, // 18 Integer::parseInt  
 (age) -> { return age + 10; });  
}

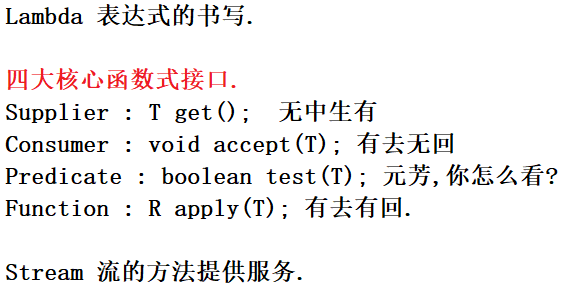


# 4. Stream 流 : (重点)

Stream 流 : 处理集合/数组中的数据. 是内存与内存之间的交互. 在 Stream 流中有大量的方法都是使用 `函数式接口` 作为方法的参数列表.

IO 流 : 硬盘与内存之间的数据交互.





## 1 传统循环遍历的弊端 :

list.add("张无忌");

list.add("周芷若");

list.add("赵敏");

list.add("张强");

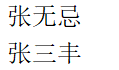
list.add("张三丰");

筛选条件如下 :

1. 首先筛选所有姓张的人；
2. 然后筛选名字有三个字的人；
3. 最后进行对结果进行打印输出。

**代码实现 :**

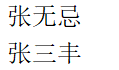
public class Test1 {  
 public static void main(String[] args) {  
  
 *// 1. 定义一个集合* ArrayList<String> list = new ArrayList<>();  
  
 *// 2. 添加数据* list.add("张无忌");  
 list.add("周芷若");  
 list.add("赵敏");  
 list.add("张强");  
 list.add("张三丰");  
  
 */\*  
 筛选条件如下 :  
 1. 首先筛选所有姓张的人；  
 2. 然后筛选名字有三个字的人；  
 3. 最后进行对结果进行打印输出。  
 \*/  
  
 // 1. 首先筛选所有姓张的人；* ArrayList<String> listZhang = new ArrayList<>();  
 *// 遍历集合* for (String name : list) {  
 *// 判断* if (name.startsWith("张")) {  
 listZhang.add(name);  
 }  
 }  
  
 *// 2. 然后筛选名字有三个字的人；* ArrayList<String> listThree = new ArrayList<>();  
 for (String name : listZhang) {  
 *// 判断* if (name.length() == 3) {  
 listThree.add(name);  
 }  
 }  
  
 *// 3. 最后进行对结果进行打印输出。* for (String name : listThree) {  
 System.*out*.println(name);  
 }  
 }  
}



## 2 Stream流的更优写法 :

**代码实现 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 *// Stream 流提供对集合中元素操作的快捷方式 :  
 // 1. 定义一个集合* ArrayList<String> list = new ArrayList<>();  
  
 *// 2. 添加数据* list.add("张无忌");  
 list.add("周芷若");  
 list.add("赵敏");  
 list.add("张强");  
 list.add("张三丰");  
  
 */\*  
 筛选条件如下 :  
 1. 首先筛选所有姓张的人；  
 2. 然后筛选名字有三个字的人；  
 3. 最后进行对结果进行打印输出。  
 \*/  
 // 步骤一 : 通过集合对象获取 `流对象`.  
 // 步骤二 : 实现对应方法的调用, 操作集合中元素数据.* list.stream()  
 .filter((name) -> { return name.startsWith("张"); }) *// 首先筛选所有姓张的人；* .filter((name) -> { return name.length() == 3; }) *// 然后筛选名字有三个字的人；* .forEach((name) -> { System.*out*.println(name); }); *// 最后进行对结果进行打印输出。* }  
}

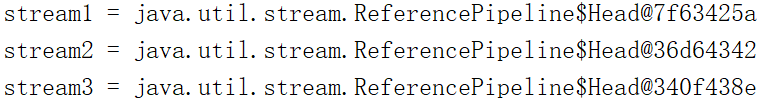


## 3 获取流 :

### 1 根据Collection获取流 :

**使用 Collection 接口定义的stream方法**

public class Test3 {  
 public static void main(String[] args) {  
   
 *// Collection 单列集合* Collection<String> c = new ArrayList<>();  
  
 List<String> list = new ArrayList<>();  
  
 Set<String> set = new HashSet<>();  
  
 Stream<String> stream1 = c.stream();  
 Stream<String> stream2 = list.stream();  
 Stream<String> stream3 = set.stream();  
 System.*out*.println("stream1 = " + stream1);  
 System.*out*.println("stream2 = " + stream2);  
 System.*out*.println("stream3 = " + stream3);  
 }  
}



### 2 根据Map获取流 :

**先获取 collection 集合, 再获取 stream 对象**

public class Test4 {  
 public static void main(String[] args) {  
  
 *// Map 双列集合* Map<String, Integer> map = new HashMap<>();  
  
 *// 1. 先获取单列集合  
 // 1.1 keySet* Set<String> keySet = map.keySet();  
 keySet.stream();  
  
 *// 2. values* Collection<Integer> values = map.values();  
 values.stream();  
  
 *// 3. entrySet* Set<Map.Entry<String, Integer>> entrySet = map.entrySet();  
 entrySet.stream();  
 }  
}

### 3 根据数组获取流

**使用 Stream 类定义的 of 静态方法**

public class Test5 {  
 public static void main(String[] args) {  
  
 int[] arr = { 10, 20, 30, 40, 50 };  
  
 *// Stream 类的静态方法 of(T...)* Stream<int[]> stream = Stream.*of*(arr);  
 System.*out*.println("stream = " + stream);  
 }  
}

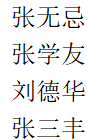
## 4 Stream 常用方法 :



### 1 逐一处理 : forEach

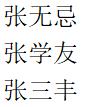
Stream<String> stream = Stream.*of*("张无忌", "张学友", "刘德华", "张三丰");

public class Test1 {  
 public static void main(String[] args) {  
  
 *// forEach(Consumer<T>) 消费型接口. (特点: 有去无回)* Stream<String> stream = Stream.*of*("张无忌", "张学友", "刘德华", "张三丰");  
  
 *// 方式一 : Lambda 实现  
 // stream.forEach((name) -> { System.out.println(name); });  
  
 // 方式二 : 方法引用* stream.forEach(System.*out*::println);  
 }  
}



### 2 过滤 : filter

public class Test2 {  
 public static void main(String[] args) {  
  
 Stream<String> stream = Stream.*of*("张无忌", "张学友", "刘德华", "张三丰");  
  
 *// 需求 : 获取流中, 所有姓张的元素  
 // filter(Predicate<T>) 特点: 元芳, 你怎么看???  
 // Stream<String> stream2 = stream.filter((name) -> { return name.startsWith("张"); });  
 // stream2.forEach(System.out::println);  
  
 // System.out.println(stream);  
 // System.out.println(stream2);  
 // stream.filter((name) -> { return name.startsWith("张"); }).forEach(System.out::println);* stream.filter(name -> name.startsWith("张")).forEach(System.*out*::println);  
 }  
}



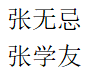
### 3 统计个数 : count

public class Test3 {  
 public static void main(String[] args) {  
  
 Stream<String> stream = Stream.*of*("张无忌", "张学友", "刘德华", "张三丰");  
  
 long count = stream.filter(name -> name.startsWith("张")).count();  
 System.*out*.println("count = " + count);  
 }  
}



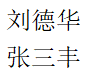
### 4 取出前几个 : limit

public class Test4 {  
 public static void main(String[] args) {  
  
 Stream<String> stream = Stream.*of*("张无忌", "张学友", "刘德华", "张三丰");  
  
 stream.filter(name -> name.startsWith("张"))  
 .limit(2)  
 .forEach(System.*out*::println);  
 }  
}



### 5 跳过前几个 : skip

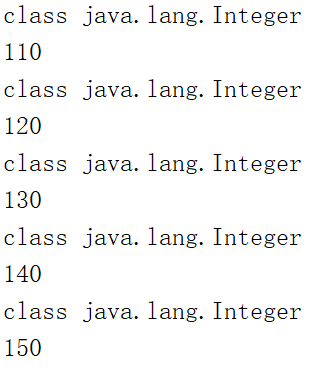
public class Test5 {  
 public static void main(String[] args) {  
  
 Stream<String> stream = Stream.*of*("张无忌", "张学友", "刘德华", "张三丰");  
  
 stream.filter(name -> name.startsWith("张"))  
 .skip(2)  
 .forEach(System.*out*::println);  
 }  
}



### 6 映射 : map

Stream<String> stream = Stream.*of*("10", "20", "30", "40", "50");

public class Test6 {  
 public static void main(String[] args) {  
  
 Stream<String> stream = Stream.*of*("10", "20", "30", "40", "50");  
  
 *// map(Function<T, R>) 元素类型决定了 T 类型. 传入Lambda表达式返回的结果就是 R 类型.  
 // 说明 : 流中的每一个元素都会依次来执行传递 Lambda 表达式. 让每一个元素都映射到 Lambda 表达式上.  
 // stream.map(Lambda表达式);  
 // stream.map((str) -> { return Integer.parseInt(str); }).forEach(System.out::println);  
  
 /\*  
 stream.map((str) -> { return Integer.parseInt(str); }).forEach((number) -> {  
 System.out.println(number.getClass());  
 System.out.println(number + 100);  
 });  
 \*/  
  
 // 方法引用 :* stream.map(Integer::*parseInt*).forEach((number) -> {  
 System.*out*.println(number.getClass());  
 System.*out*.println(number + 100);  
 });  
 }  
}

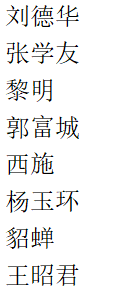


### 7 组合 : concat

Stream<String> stream1 = Stream.*of*("刘德华", "张学友", "黎明", "郭富城");  
Stream<String> stream2 = Stream.*of*("西施", "杨玉环", "貂蝉", "王昭君");

**拼接实现 :**

public class Test7 {  
 public static void main(String[] args) {  
  
 Stream<String> stream1 = Stream.*of*("刘德华", "张学友", "黎明", "郭富城");  
 Stream<String> stream2 = Stream.*of*("西施", "杨玉环", "貂蝉", "王昭君");  
  
 *// 需求 : 能够将两个流对象, 合并为一个流对象实现操作.* Stream<String> stream = Stream.*concat*(stream1, stream2);  
  
 stream.forEach(System.*out*::println);  
 }  
}



## 5 练习 : 集合元素处理 (传统方式)

需求 : 现在有两个ArrayList集合存储队伍当中的多个成员姓名，要求使用传统的for循环（或增强for循环）依次进行以下若干操作步骤：

1. 第一个队伍只要名字为3个字的成员姓名；
2. 第一个队伍筛选之后只要前3个人；
3. 第二个队伍只要姓张的成员姓名；
4. 第二个队伍筛选之后不要前2个人；
5. 将两个队伍合并为一个队伍；
6. 根据姓名创建Person对象；
7. 打印整个队伍的Person对象信息。

**集合代码如下 :**

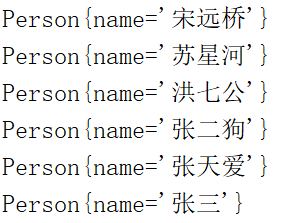
List<String> one = new ArrayList<>();  
one.add("迪丽热巴");  
one.add("宋远桥");  
one.add("苏星河");  
one.add("老子");  
one.add("庄子");  
one.add("孙子");  
one.add("洪七公");  
  
List<String> two = new ArrayList<>();  
two.add("古力娜扎");  
two.add("张无忌");  
two.add("张三丰");  
two.add("赵丽颖");  
two.add("张二狗");  
two.add("张天爱");  
two.add("张三");

**Person 代码如下 :**

public class Person {  
 private String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

**传统方法代码操作实现 :**

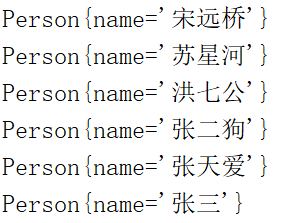
public class Test1 {  
 public static void main(String[] args) {  
  
 List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");  
  
 List<String> two = new ArrayList<>();  
 two.add("古力娜扎");  
 two.add("张无忌");  
 two.add("张三丰");  
 two.add("赵丽颖");  
 two.add("张二狗");  
 two.add("张天爱");  
 two.add("张三");  
  
 *// 1. 第一个队伍只要名字为3个字的成员姓名；* ArrayList<String> oneA = new ArrayList<>();  
 for (String name : one) {  
 if (name.length() == 3) {  
 oneA.add(name);  
 }  
 }  
  
 *// 2. 第一个队伍筛选之后只要前3个人；* ArrayList<String> oneB = new ArrayList<>();  
 for (int i = 0; i < 3; i++) {  
 String name = oneA.get(i);  
 oneB.add(name);  
 }  
  
 *// 3. 第二个队伍只要姓张的成员姓名；* ArrayList<String> twoA = new ArrayList<>();  
 for (String name : two) {  
 if (name.startsWith("张")) {  
 twoA.add(name);  
 }  
 }  
  
 *// 4. 第二个队伍筛选之后不要前2个人；* ArrayList<String> twoB = new ArrayList<>();  
 for (int i = 2; i < twoA.size(); i++) {  
 String name = twoA.get(i);  
 twoB.add(name);  
 }  
  
 *// 5. 将两个队伍合并为一个队伍；* oneB.addAll(twoB);  
  
 *// System.out.println("oneB = " + oneB);  
  
 // 6. 根据姓名创建Person对象；* ArrayList<Person> people = new ArrayList<>();  
 for (String name : oneB) {  
 *// 根据名字创建对应的 Person 对象* Person p = new Person(name);  
 people.add(p);  
 }  
  
 *// 7. 打印整个队伍的Person对象信息。* for (Person person : people) {  
 System.*out*.println(person);  
 }  
 }  
}



## 6 练习 : 集合元素处理 (Stream流方式)

**Stream 流操作方式代码实现 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");  
  
 List<String> two = new ArrayList<>();  
 two.add("古力娜扎");  
 two.add("张无忌");  
 two.add("张三丰");  
 two.add("赵丽颖");  
 two.add("张二狗");  
 two.add("张天爱");  
 two.add("张三");  
  
 *// 1. 第一个队伍只要名字为3个字的成员姓名；  
 // one.stream().filter((name) -> { return name.length() == 3; });  
  
 // 2. 第一个队伍筛选之后只要前3个人；* Stream<String> streamA = one.stream().filter((name) -> { return name.length() == 3; }).limit(3);  
  
 *// 3. 第二个队伍只要姓张的成员姓名；  
 // two.stream().filter((name) -> { return name.startsWith("张"); });  
  
 // 4. 第二个队伍筛选之后不要前2个人；* Stream<String> streamB = two.stream().filter((name) -> { return name.startsWith("张"); }).skip(2);  
  
 *// 5. 将两个队伍合并为一个队伍；* Stream<String> stream = Stream.*concat*(streamA, streamB);  
  
 *// 6. 根据姓名创建Person对象；  
 // stream.map((name) -> { return new Person(name); });  
  
 // 构造方法引用 : Person (String)  
 // stream.map(Person::new);  
  
 // 7. 打印整个队伍的Person对象信息。  
 // stream.map((name) -> { return new Person(name); }).forEach((p) -> { System.out.println(p);});* stream.map(Person::new).forEach(System.*out*::println);  
 }  
}



## 7 并发流 : (多线程处理集合中的数据)

### 1 Stream流转换为并发流 : paralell方法

Stream<Integer> stream = Stream.*of*(10, 20, 30, 40, 50);

public class Test1 {  
 public static void main(String[] args) {  
  
 Stream<Integer> stream = Stream.*of*(10, 20, 30, 40, 50);  
  
 *// 底层就是 `单线程` 操作流中的元素  
 // stream.forEach(System.out::println);  
  
 // 需要 : 希望底层使用 `多线程` 操作流中的元素  
 // Stream 流提供的 parallel();* stream.parallel().forEach(System.*out*::println);   
 }  
}



### 2 集合对象直接获取并发流 : paralellStream 方法

public class Test1 {  
 public static void main(String[] args) {  
  
 *// 集合对象直接获取 `并发流` 对象呢 ??? Collection 集合提供的 parallelStream();* ArrayList<Integer> list = new ArrayList<>();  
 Collections.*addAll*(list, 10, 20, 30, 40, 50);  
  
 list.parallelStream().forEach(System.*out*::println);  
 }  
}



## 8 收集 Stream 结果 : collect 方法

### 1 收集到集合中 : collectors调用toList 方法

List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");

*// 需求 : 将数据实现过滤, 然后收集*

public class Test1 {  
 public static void main(String[] args) {  
  
 List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");  
  
 *// 需求 : 获取三个字的元素, 重新收集到一个新的 list 集合中.* List<String> list = one.stream().filter(name -> name.length() == 3).collect(Collectors.*toList*());  
 System.*out*.println(list);  
  
 *// set 集合中* Set<String> set = one.stream().filter(name -> name.length() == 3).collect(Collectors.*toSet*());  
 System.*out*.println(set);   
 }  
}





### 2 收集到数组中 : stream调用toArray方法

List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");

*// 需求 : 将数据实现过滤, 然后收集, 收集到数组中*

public class Test1 {  
 public static void main(String[] args) {  
  
 List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");

*// 数组* Object[] objects = one.stream().filter(name -> name.length() == 3).toArray();  
 System.*out*.println("Arrays.toString(objects) = " + Arrays.*toString*(objects));  
   
 }  
}



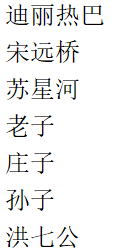
### 3 扩展 : 解决泛型数组问题 : 传递数组的构造引用

List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");

*// toArray 可以指定数组类型的构造器 (数组的构造引用)*

public class Test1 {  
 public static void main(String[] args) {  
  
 List<String> one = new ArrayList<>();  
 one.add("迪丽热巴");  
 one.add("宋远桥");  
 one.add("苏星河");  
 one.add("老子");  
 one.add("庄子");  
 one.add("孙子");  
 one.add("洪七公");

*// 泛型数组  
 // R apply(int value);  
 // String[] strings = one.stream().filter(name -> name.length() == 3).toArray((len) -> { return new String[len]; });* String[] strings = one.stream().filter(name -> name.length() == 3).toArray(String[]::new); *// generator 数组的生成器* System.*out*.println("Arrays.toString(strings) = " + Arrays.*toString*(strings));  
 }  
}



## 9 练习 : 将数组元素添加到集合中 (of+collect)

String[] names = {"张三", "李四", "王五", "赵六", "田七"};

public class Test3 {  
 public static void main(String[] args) {  
  
 String[] names = {"张三", "李四", "王五", "赵六", "田七"};  
  
 *// 数组 -> 集合  
 /\*  
 ArrayList<String> list = new ArrayList<>();  
 for (String name : names) {  
 list.add(name);  
 }  
 System.out.println(list);  
 \*/  
  
 // Stream.of(数组)* List<String> list = Stream.*of*(names).collect(Collectors.*toList*());  
 System.*out*.println(list);  
 }  
}



理解说明 :

1. 能够理解流与集合相比的优点.

说明 : 流对象可以直接对集合元素实现 `循环的中间操作` 和最后的 `收集或输出` 操作.

2. 能够通过集合、映射或数组获取流

单列集合 : stream();

双列集合 : 1. 转换单列集合 (keySet, values, entrySet) 2. 调用 stream() 方法.

数组 : Stream 静态方法 of(T …); 可变参数底层就是数组.

3. 能够掌握常用的流操作

方法: filter, map, limit, skip, concat, foreach, count, collect…

4. 能够将流中的内容收集到集合中

收集 : collect 集合收集工具类 : collectors -> toList(); toSet();

5. 能够将流中的内容收集到数组中

数组 : 不需要使用 collect 方法, 直接调用 toArray 方法. 同时也可以传递一个 `generator` 数组构造器, 数组生成器.

# 10. Junit 单元测试 :

**Junit** -> Java Unit Java单元测试

Junit 是一个第三方框架 : 框架可以理解为功能的集合体.

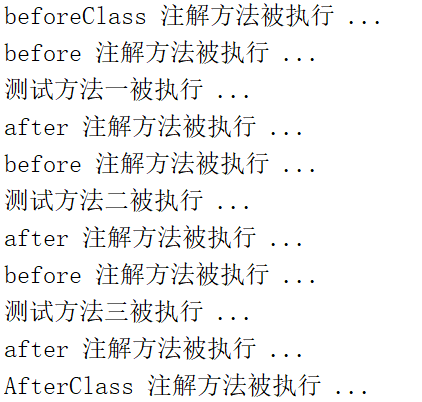
Junit 是用来测试 Java代码方法的. 类似于 main 方法, 可以让一个方法独立被执行, 不需要在代码中使用 main 方法调用.

**IDEA 集成开发环境 :** 该开发工具已经集成了 Junit 单元测试, 直接在工具中添加即可. 不需要去 Junit 官网下载.

Jar : Java 字节码文件的压缩包.

@Test 需要被独立测试方法使用 Test 测试注解. 可以单独实现方法的测试.  
 @Before 在每一个测试方法执行之前被执行.  
 @After 在每一个测试方法执行之后被执行.  
  
 @BeforeClass 在类测试之前被执行. (最早, 仅执行一次)  
 @AfterClass 在类测试之后被执行. (最晚, 仅执行一次)

public class Test1 {  
  
 *// 在测试类执行之前被执行 ... (最早, 仅执行一次)* @BeforeClass  
 public static void beforeClass() {  
 System.*out*.println("beforeClass 注解方法被执行 ...");  
 }  
  
 *// before 注解修饰方法, 在每一个 test 方法执行之前先被执行* @Before  
 public void before() {  
 System.*out*.println("before 注解方法被执行 ...");  
 }  
  
 @Test  
 public void test1() {  
 System.*out*.println("测试方法一被执行 ...");  
 }  
  
 @Test  
 public void test2() {  
 System.*out*.println("测试方法二被执行 ...");  
 }  
  
 @Test  
 public void test3() {  
 System.*out*.println("测试方法三被执行 ...");  
 }  
  
 @After  
 public void after() {  
 System.*out*.println("after 注解方法被执行 ...");  
 }  
  
 *// 在本类结束之前被执行. (最后, 仅执行一次)* @AfterClass  
 public static void AfterClass() {  
 System.*out*.println("AfterClass 注解方法被执行 ...");  
 }  
}



**Calculator测试类 :**

public class Calculator {  
  
 *// 行为 :* public static int getSum(int num1, int num2) {  
 return num1 + num2;  
 }  
  
 public static int getDivide(int num1, int num2) {  
  
 *// double result = (double) 10.0 / 0;  
 // double result = (double) num1 / num2; 结果为: Infinity 无穷大* return num1 / num2;  
 }  
}

**代码测试 :**

public class Test2 {  
 @Test  
 public void test1() {  
 int sum = Calculator.*getSum*(10, 20);  
 System.*out*.println("sum = " + sum);  
 }  
  
 @Test  
 public void test2() {  
 int divide = Calculator.*getDivide*(10, 0);  
 System.*out*.println("divide = " + divide);  
 }  
}

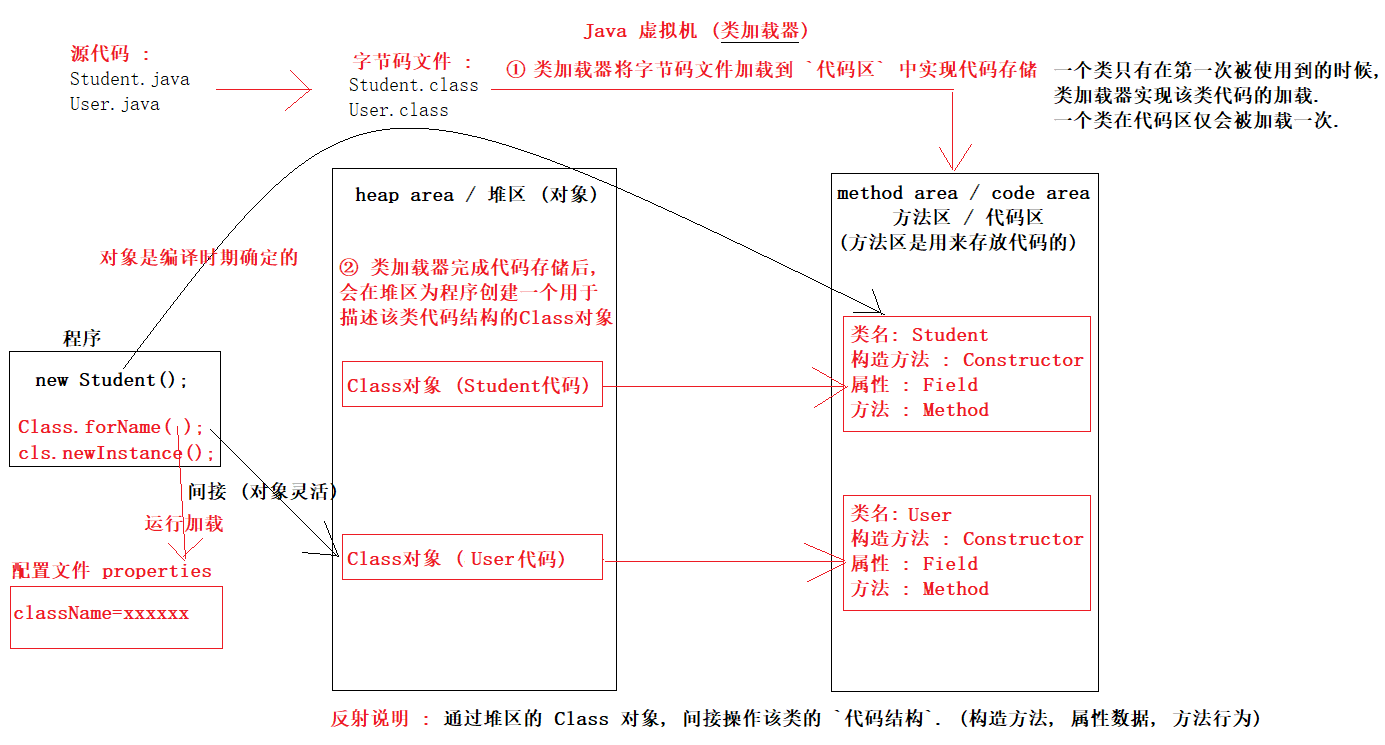


# 11. 反射 Reflection : (了解)

## 1 反射的概念 :

**应用场景 :** IDEA集成开发工具, 框架底层使用. 通用类.

**说明 :** 在自己正常逻辑代码编写中, 很少会使用到反射的技术. 学习反射的目的就是为了更好的理解框架的底层实现. 为后期的学习做好一个铺垫工作.

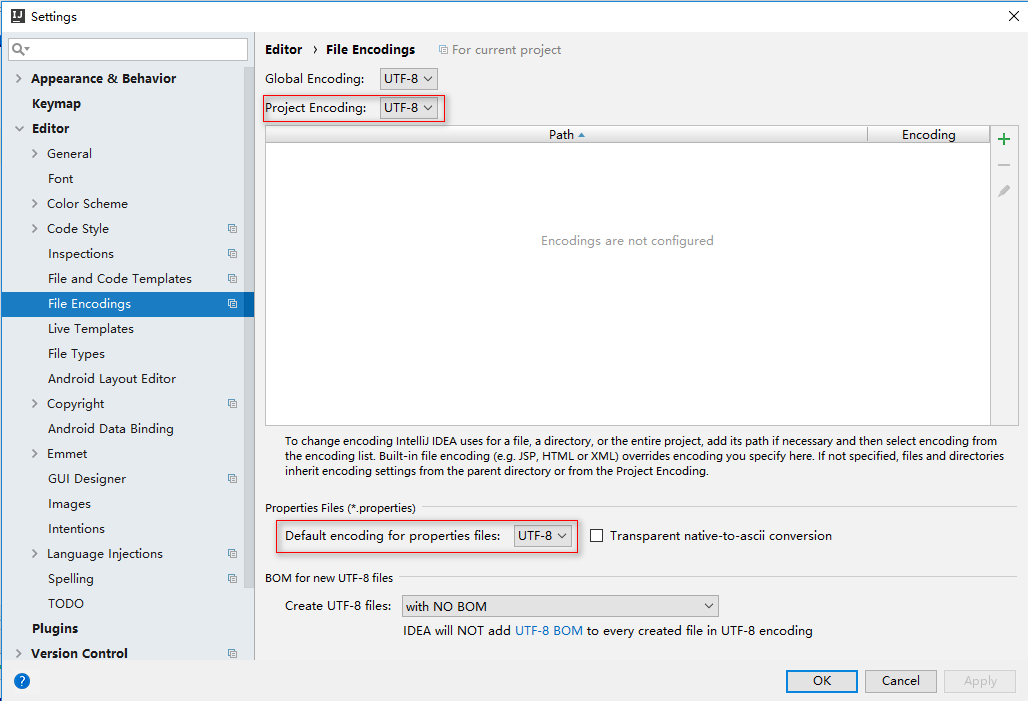


**类加载器的作用 :**

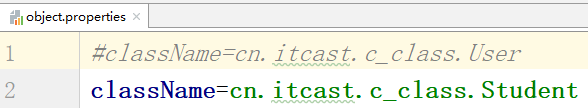
1. 在第一次程序使用到某类时, 类加载器将该类代码加载到 `代码区` 中实现代码的存储.

2. 类加载器完成代码存储后, 会在堆区为程序创建一个用于描述该类代码结构的 Class 对象.

**反射初体验 :**

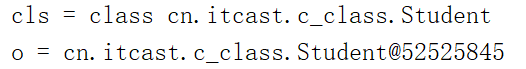


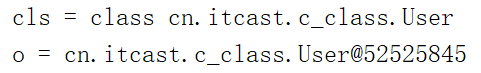
**配置文件 :**



**测试类 :**

@Test  
public void test1() throws Exception {  
  
 Properties prop = new Properties();  
 prop.load(new FileReader("object.properties"));  
 String className = prop.getProperty("className");  
  
 *// 方式一 : Class.forName(类的全限定名称); 包名 + 类名  
 // 使用场景 : 根据配置文件中的 `全限定名称` 来获取对应 Class 对象.* Class<?> cls = Class.*forName*(className);  
 System.*out*.println("cls = " + cls);  
 Object o = cls.newInstance();  
 System.*out*.println("o = " + o);  
}

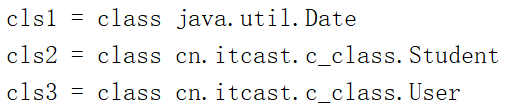




## 2 获取Class对象的三种方式 :

@Test  
public void test1() throws Exception {  
  
 *// 方式一 : Class.forName(类的全限定名称); 包名 + 类名  
 // 使用场景 : 根据配置文件中的 `全限定名称` 来获取对应 Class 对象.* Class<?> cls = Class.*forName*("cn.itcast.c\_class.Student");  
 System.*out*.println("cls = " + cls);   
}

@Test  
public void test2() {  
  
 // 方式二 : 类名.class 属性  
 // 使用场景 : 确定调用的唯一方法. `方法名 + 参数列表`. sum(float, float)  
 // Java 语言中, 只要是 `类型`, 就拥有 class 属性.  
 // sum(float.class, float.class);  
  
 // ? extends Object  
 Class<?> cls1 = Date.class;  
 Class<?> cls2 = Student.class;  
 Class<?> cls3 = User.class;  
 System.out.println("cls1 = " + cls1);  
 System.out.println("cls2 = " + cls2);  
 System.out.println("cls3 = " + cls3);  
}

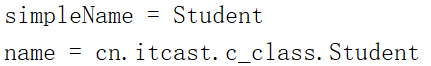


@Test  
public void test3() {  
  
 *// 方式三 : 对象名.getClass(); 方法.  
 // 常用场景 : 在方法内部确定参数的具体类型* show(new User());  
}  
  
public void show(Object obj) {  
 Class<?> cls = obj.getClass(); *// getClass() 可以确定对象的唯一类型.* System.*out*.println("cls = " + cls);  
}



## 3 获取Class对象的信息 : (名称)

@Test  
public void test1() throws Exception {  
  
 Class<?> cls = Class.*forName*("cn.itcast.c\_class.Student");  
  
 *// 名称 : simpleName = Student* String simpleName = cls.getSimpleName();  
 System.*out*.println("simpleName = " + simpleName);  
  
 *// name = cn.itcast.c\_class.Student* String name = cls.getName();  
 System.*out*.println("name = " + name);  
}



## 4 获取Class对象的Constructor信息 :

public class Student {  
 private String name;  
 private int age;  
  
 *// 私有构造方法* private Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student(String name) {  
 this.name = name;  
 }  
  
 public Student() {  
  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

**公开无参构造方法 :**

@Test  
public void test2() throws Exception {  
  
 // 1. 获取用于表示 Student 类的 Class 对象.  
 Class<?> cls = Class.forName("cn.itcast.d\_constructor.Student");  
  
 // 2. 根据 cls 对象创建 `cls对象表示的真实对象`  
 // Object o = cls.newInstance(); 已过时, 不推荐使用.  
 Object o = cls.getDeclaredConstructor().newInstance(); // 替换方案. 说明: 如果无参的构造方法为私有, 创建失败.(不可访问)  
 System.out.println("o = " + o);  
}



@Test  
public void test4() throws Exception {  
  
 // 1. 获取用于表示 Student 类的 Class 对象.  
 Class<?> cls = Class.forName("cn.itcast.d\_constructor.Student");  
  
 // 2. 根据 cls 对象创建 `cls对象表示的真实对象`  
 // Constructor<?> constructor = cls.getConstructor(new Class[]{}); // 数组大括号中不传递参数, 表示无参  
 Constructor<?> constructor = cls.getConstructor(); // 可变参数无参时, 可以什么都不传递.  
  
 // 3. 创建真实对象  
 // Object o = constructor.newInstance(new Object[]{});  
 Object o = constructor.newInstance();  
 System.out.println("o = " + o);  
}



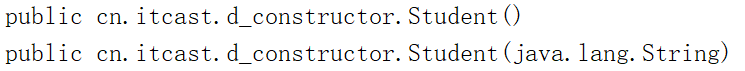
**私有方法 :**

// 私有构造方法  
@Test  
public void test3() throws Exception {  
  
 // 1. 获取对应 Class 对象  
 Class<?> cls = Class.forName("cn.itcast.d\_constructor.Student");  
  
 // 2. 调用 cls 对象 getDeclaredConstructor  
 // Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)  
 // 请问 : Class ... 可变参数, 底层就是 Class 数组. new Class[]{} 创建了一个 Class 数组.  
 // 向获取一个构造方法, 构造方法可能有参数, 参数类型是什么 ???  
 Constructor<?> constructor = cls.getDeclaredConstructor(new Class[]{String.class, int.class}); // 形参列表  
  
 // 3. 如果获取对象的为私有, 必须先进行 `暴力反射` / `暴力访问`.  
 constructor.setAccessible(true); // 设置访问权限, 允许访问.  
  
 // 4. 使用 constructor 对象, 调用 newInstance(...) 方法, 创建真实对象  
 // T newInstance(Object ... initargs);  
 // 请问 : Object ... 可变参数, 底层就是 Object 数组. new Object[]{} 创建了一个 Object 数组  
 Object o = constructor.newInstance(new Object[]{ "渣渣辉", 48 }); // 实参列表  
 System.out.println("o = " + o);  
}



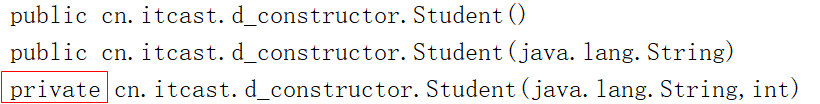
**所有公开构造方法 :**

@Test  
public void test5() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.d\_constructor.Student");  
  
 *// 2. 获取所有的公开构造方法* Constructor<?>[] constructors = cls.getConstructors();  
 for (Constructor<?> constructor : constructors) {  
 System.*out*.println(constructor);  
 }  
}



**所有公开和私有构造方法 :**

@Test  
public void test6() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.d\_constructor.Student");  
  
 *// 2. 获取所有的公开构造方法* Constructor<?>[] constructors = cls.getDeclaredConstructors();  
 for (Constructor<?> constructor : constructors) {  
 System.*out*.println(constructor);  
 }  
}

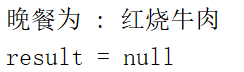


## 5 获取Class对象的Method信息 :

public class Student {  
 private String name;  
 private int age;  
  
 *// 私有构造方法* private Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student(String name) {  
 this.name = name;  
 }  
  
 public Student() {  
  
 }  
  
 *// 行为 : 静态方法* public static void eat(String food) {  
 System.*out*.println("晚餐为 : " + food);  
 }  
  
 *// 公开方法对象方法* public String study() {  
 System.*out*.println("好好学习, 天天向上.");  
 return "考试满分, 耶!";  
 }  
  
 *// 私有方法* private boolean fallInLove(String boy, String girl) {  
 System.*out*.println(boy + " loves the " + girl);  
 return true;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

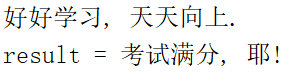
**公开静态方法 :**

@Test  
public void test1() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.e\_method.Student");  
  
 *// 2. 通过 cls 对象获取对应的方法  
 // Method getMethod(String name, Class<?>... parameterTypes)* Method eat = cls.getMethod("eat", new Class[]{String.class});  
 *// cls.getMethod("eat", String.class);  
  
 // 3. 调用该方法  
 // Object invoke(Object obj, Object... args)  
 // Object 表示对象.* Object result = eat.invoke(null, new Object[]{ "红烧牛肉 "});  
 System.*out*.println("result = " + result);  
}



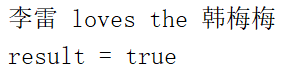
**公开对象方法 :**

@Test  
public void test2() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.e\_method.Student");  
 Object obj = cls.getDeclaredConstructor().newInstance();  
  
 *// 2. 通过 cls 对象获取对应的方法  
 // Method getMethod(String name, Class<?>... parameterTypes)* Method study = cls.getMethod("study", new Class[]{});  
  
 *// 3. 调用该方法  
 // Object invoke(Object obj, Object... args)  
 // Object 表示对象.* Object result = study.invoke(obj, new Object[]{ });  
 System.*out*.println("result = " + result);  
}



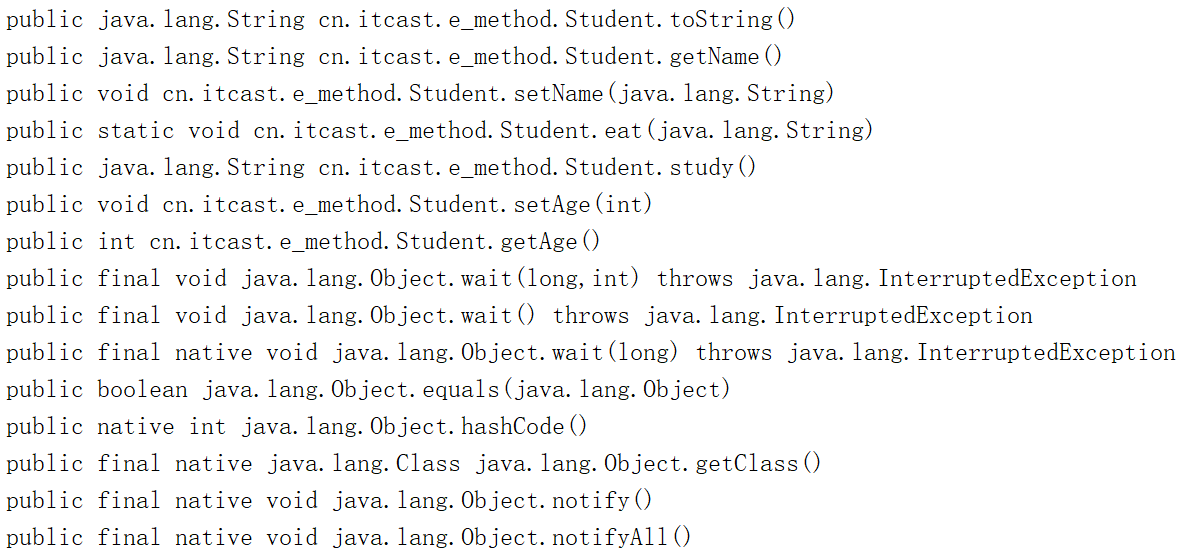
**私有方法 :**

@Test  
public void test3() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.e\_method.Student");  
 Object obj = cls.getDeclaredConstructor().newInstance();  
  
 *// 2. 通过 cls 对象获取对应的方法  
 // Method getDeclaredMethod(String name, Class<?>... parameterTypes)* Method love = cls.getDeclaredMethod("fallInLove", new Class[]{ String.class, String.class });  
  
 *// 3. 暴力反射* love.setAccessible(true);  
  
 *// 4. 调用该方法  
 // Object invoke(Object obj, Object... args)  
 // Object 表示对象.* Object result = love.invoke(obj, new Object[]{ "李雷", "韩梅梅" });  
 System.*out*.println("result = " + result);  
}



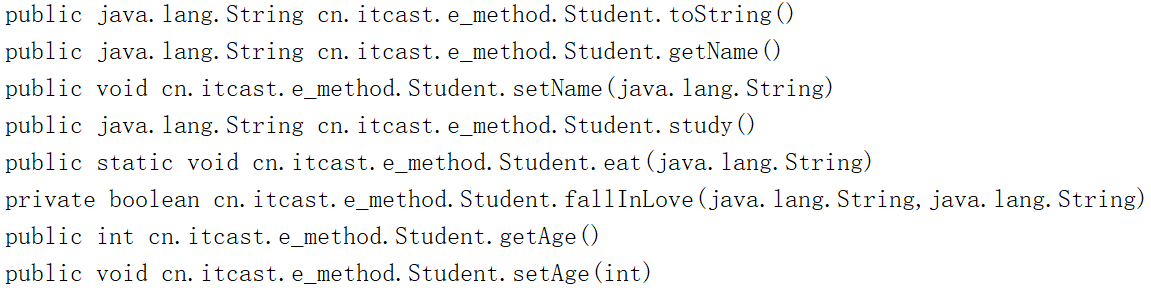
**所有公开方法 :**

@Test  
public void test4() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.e\_method.Student");  
  
 *// 2. 获取所有公开方法, (包含继承的)* Method[] methods = cls.getMethods();  
 for (Method method : methods) {  
 System.*out*.println(method);  
 }  
}



**所有公开和私有方法 :**

@Test  
public void test5() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.e\_method.Student");  
  
 *// 2. 获取所有声明方法, (不包含继承的, 但包含本类私有的)* Method[] methods = cls.getDeclaredMethods();  
 for (Method method : methods) {  
 System.*out*.println(method);  
 }  
}

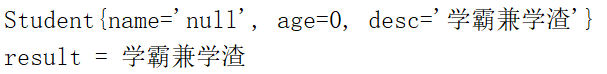


## 6 获取Class对象的Field信息 :

public class Student {  
 private String name;  
 private int age;  
  
 *// 公开的* public String desc; *// description 描述  
  
 // 私有构造方法* private Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Student(String name) {  
 this.name = name;  
 }  
  
 public Student() {  
  
 }  
  
 *// 行为 : 静态方法* public static void eat(String food) {  
 System.*out*.println("晚餐为 : " + food);  
 }  
  
 *// 公开方法对象方法* public String study() {  
 System.*out*.println("好好学习, 天天向上.");  
 return "考试满分, 耶!";  
 }  
  
 *// 私有方法* private boolean fallInLove(String boy, String girl) {  
 System.*out*.println(boy + " loves the " + girl);  
 return true;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", desc='" + desc + '\'' +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**公开属性 :**

@Test  
public void test1() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.f\_field.Student");  
 Object obj = cls.getDeclaredConstructor().newInstance();  
  
 *// 2. 获取属性* Field desc = cls.getField("desc");  
  
 *// 3. 设置数值* desc.set(obj, "学霸兼学渣");  
  
 System.*out*.println(obj);  
  
 Object result = desc.get(obj);  
 System.*out*.println("result = " + result);  
}



**私有属性 :**

@Test  
public void test2() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.f\_field.Student");  
 Object obj = cls.getDeclaredConstructor().newInstance();  
  
 *// 2. 获取属性* Field name = cls.getDeclaredField("name");  
  
 *// 暴力反射* name.setAccessible(true);  
  
 *// 3. 设置数值* name.set(obj, "渣渣辉");  
  
 System.*out*.println(obj);  
}



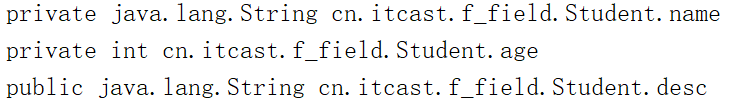
**所有公开属性 :**

@Test  
public void test3() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.f\_field.Student");  
  
 *// 2. 所有属性对象* Field[] fields = cls.getFields();  
 for (Field field : fields) {  
 System.*out*.println(field);  
 }  
}



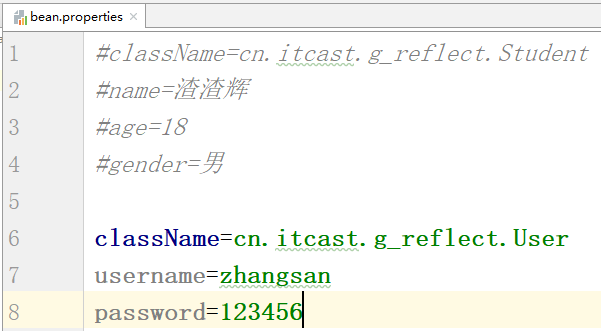
**所有公开和私有属性 :**

@Test  
public void test4() throws Exception {  
  
 *// 1. 获取 Class 对象* Class<?> cls = Class.*forName*("cn.itcast.f\_field.Student");  
  
 *// 2. 所有属性对象* Field[] fields = cls.getDeclaredFields();  
 for (Field field : fields) {  
 System.*out*.println(field);  
 }  
}



## 7 反射案例 :

**配置文件 : properties 注意: key 不能重复 (简单配置)**



**JavaBean 类 : User, Student**

Bean 表示一个标准的 Java 类.

1. 该类必须要提供一个无参的公开构造方法.

2. 该类的属性私有化, 必须要提供标准的 getter / setter 方法.

3, 该类必须是公开的.

public class Student {  
 private String name;  
 private int age;  
 private char gender;  
  
 public Student(String name, int age, char gender) {  
 this.name = name;  
 this.age = age;  
 this.gender = gender;  
 }  
  
 public Student() {  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", gender=" + gender +  
 '}';  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public char getGender() {  
 return gender;  
 }  
  
 public void setGender(char gender) {  
 this.gender = gender;  
 }  
}

public class User {  
 private String username;  
 private String password;  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "username='" + username + '\'' +  
 ", password='" + password + '\'' +  
 '}';  
 }  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
}

**核心类 :**

// 作用 : 专门根据提供的配置创建对应的 JavaBean 对象.  
public class BeanCreate {  
  
 // 属性  
 private static Properties prop;  
   
 // 步骤一 : 加载配置文件. (静态代码块)  
 // 静态代码块是在类加载器加载该类的最后一步执行的.  
 // 特点 : 执行的时间非常早. 仅执行一次.  
 static {  
 // System.out.println("静态代码块被执行...");  
  
 // 1. 初始化属性集对象  
 prop = new Properties();  
  
 try {  
 // 2. 加载文件数据  
 prop.load(new FileReader("bean.properties"));  
 } catch (Exception e) {  
 // 将编译时期异常转换为运行时期异常, 抛出, 终止程序继续执行.  
 throw new RuntimeException("配置文件加载失败!");  
 }  
 }  
  
 // 方法 :  
 public static Object createBean() {  
  
 // 1. 获取 className 名称对应的 `类的全限定名称`  
 String className = prop.getProperty("className");  
 // System.out.println(className); // cn.itcast.g\_reflect.User  
  
 try {  
 // 2. 根据 className 获取对应的 Class 对象  
 Class<?> cls = Class.forName(className);  
  
 // 3. 根据 cls 对象创建一个真实对象  
 Object o = cls.getDeclaredConstructor().newInstance();  
  
 // 4. 获取 prop 对象中的所有 key  
 Set<String> keys = prop.stringPropertyNames();  
  
 // 5. 遍历 keys 集合  
 for (String key : keys) {  
  
 // 5.1 排除 className 属性  
 if ("className".equals(key)) continue;  
  
 // 5.2 根据字符串属性获取对应 Field 对象.  
 Field field = cls.getDeclaredField(key);  
  
 // 暴力反射  
 field.setAccessible(true);  
  
 // 5.3 根据 key 从 prop 对象中取值  
 String value = prop.getProperty(key);  
  
 // 属性类型判断  
 if (field.getType() == char.class) {  
 char c = value.charAt(0);  
 field.set(o, c);  
 } else if (field.getType() == int.class) {  
 int i = Integer.parseInt(value);  
 field.set(o, i);  
 } else if (field.getType() == String.class) {  
 // 5.4. 使用 field 对象调用 set 方法为反射创建的 o 对象赋值  
 field.set(o, value);  
 }  
 // System.out.println(key);  
 }  
  
 return o;  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
}

**测试类 :**

public class Test {  
 public static void main(String[] args) {  
  
 Object bean = BeanCreate.*createBean*();  
 System.*out*.println("bean = " + bean);  
 }  
}





# 12. 注解 : (了解)

## 1 注解的概述 :

注解是给编译器查看的.

1. 可以决定程序如何运行. 例如 @Test

2. 注解可以实现编译阶段的指定语法检查. 例如: @functionalInterface @Override

3. 注解可以给程序提供信息.

## 2 注解的属性 :

**属性就是信息.**

public @interface Student {  
  
 /\*  
 1. 基本数据类型 : byte, short, int, long, float, double, char, boolean  
 2. 引用类型 : String, Class, Enum, Annotation  
 3. 特殊类型 : 数组类型 String[]  
 \*/  
  
 // 姓名  
 String name();  
  
 // 年龄  
 int age();  
  
 // 性别  
 char gender();  
  
 // Date date();  
}

## 3 使用自定义注解 :

**Book 注解 (value, price, authors) BookShelf 书架类使用该注解.**

**Book 注解定义 :**

public @interface Book {  
  
 *// 说明 : 如果注解仅有一个属性, 请将名称定义为 value. 因为 value 可以省略.* String value();  
  
 *// 多个属性* int price() default 998;  
  
 String[] authors();  
}

**BookShelf 类使用注解 :**

public class BookShelf {  
  
 @Book(value = "降龙十八掌", price = 100, authors = {"乔峰", "郭靖", "扫地僧"})  
 public void showBookInfo() {  
  
 String name = "";  
 int price = 0;  
 String[] authors = {};  
  
 *// 默认值* System.*out*.println("name = " + name);  
 System.*out*.println("price = " + price);  
 System.*out*.println("authors = " + Arrays.*toString*(authors));  
 }  
}

## 4 注解解析 (AnnotatedElement接口) :



public class BookShelf {  
  
 @Book(value = "降龙十八掌", price = 100, authors = {"乔峰", "郭靖", "扫地僧"})  
 public void showBookInfo() {  
  
 String name = "";  
 int price = 0;  
 String[] authors = {};  
  
 *// 如何将注解的数据实现解析 ??? (解析实现仅做了解)  
 // AnnotatedElement : 接口 (实现类: Class, Constructor, Method, Field)  
 // 一个判断注解方法 isAnnotationPresent(注解类型)  
 // 获取指定注解对象 getAnnotation(注解类型)  
  
 // 1. 获取本类的 Class 对象* Class<? extends BookShelf> cls = this.getClass();  
 try {  
 *// 2. 获取指定的方法* Method method = cls.getMethod("showBookInfo");  
 *// 3. 判断 method 对象上是否存在指定注解* if (method.isAnnotationPresent(Book.class)) {  
 *// 4. 条件成立, 获取指定注解* Book book = method.getAnnotation(Book.class);  
 *// 5. 获取 book 注解对象上的数据* name = book.value();  
 price = book.price();  
 authors = book.authors();  
 }  
  
 } catch (Exception e) {  
  
 }  
  
 *// 默认值* System.*out*.println("name = " + name);  
 System.*out*.println("price = " + price);  
 System.*out*.println("authors = " + Arrays.*toString*(authors));  
 }  
}

## 5 元注解 :

修饰注解的注解 :

@Target 注解的目标

ElementType : Type(类,接口) Method(方法) Field(属性) …

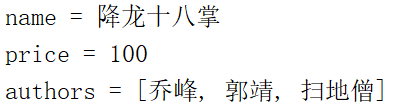
@Retention 保留策略

RetentionPolicy : SOURCE(源码) CLASS(字节码) RUNTIME(运行时)

// 元注解 : 修饰注解的注解  
@Target({ElementType.METHOD}) // 注解使用的目标位置  
  
@Retention(RetentionPolicy.RUNTIME) // 反射获取数据, 必须选择 RUNTIME  
  
public @interface Book {  
  
 // 说明 : 如果注解仅有一个属性, 请将名称定义为 value. 因为 value 可以省略.  
 String value();  
  
 // 多个属性  
 int price() default 998;  
  
 String[] authors();  
}

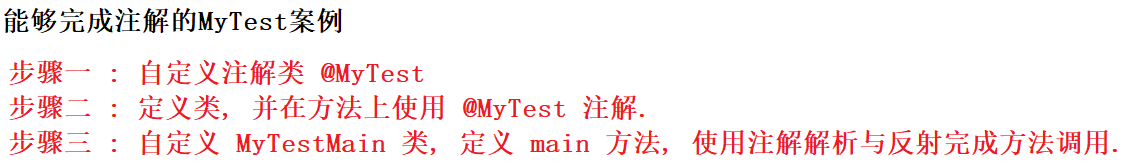
**测试类 :**

public class Test2 {  
 public static void main(String[] args) {  
  
 BookShelf bookShelf = new BookShelf();  
 bookShelf.showBookInfo();  
 }  
}





## 6 注解案例 MyTest :



**自定义注解 MyTest :**

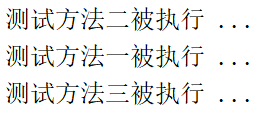
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
*// 自定义一个注解类 : MyTest*@Target(ElementType.*METHOD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface MyTest {  
  
}

**使用自定义注解 MyTest :**

public class Test1 {  
  
 @MyTest  
 public void test1() {  
 System.*out*.println("测试方法一被执行 ...");  
 }  
  
 @MyTest  
 public void test2() {  
 System.*out*.println("测试方法二被执行 ...");  
 }  
  
 @MyTest  
 public void test3() {  
 System.*out*.println("测试方法三被执行 ...");  
 }  
}

**模拟代码 : (反射执行方法)**

public class MyTestMain {  
 public static void main(String[] args) throws Exception {  
  
 // 执行测试类中被 MyTest 修饰的注解方法.  
 // 全限定名 : cn.itcast.b\_anno.Test1  
  
 // 注解解析 + 反射 method 执行方法  
  
 // 1. 获取执行类的 Class 对象  
 Class<?> cls = Class.forName("cn.itcast.b\_anno.Test1");  
 Object obj = cls.getDeclaredConstructor().newInstance();  
  
 // 2. 获取 cls 表示真实类的所有 `方法对象`  
 Method[] methods = cls.getMethods();  
  
 // 3. 遍历 methods 方法数组  
 for (Method method : methods) {  
  
 // 4. 使用 method 方法对象调用 isAnnotationPresent 方法, 判断该方法是否存在指定注解  
 if (method.isAnnotationPresent(MyTest.class)) {  
  
 // 5. 如果条件成立, 说明需要执行该方法  
 // method.invoke(obj, new Class[]{});  
 method.invoke(obj);  
 }  
 }  
 }  
}



# 13. 代理 : (了解)

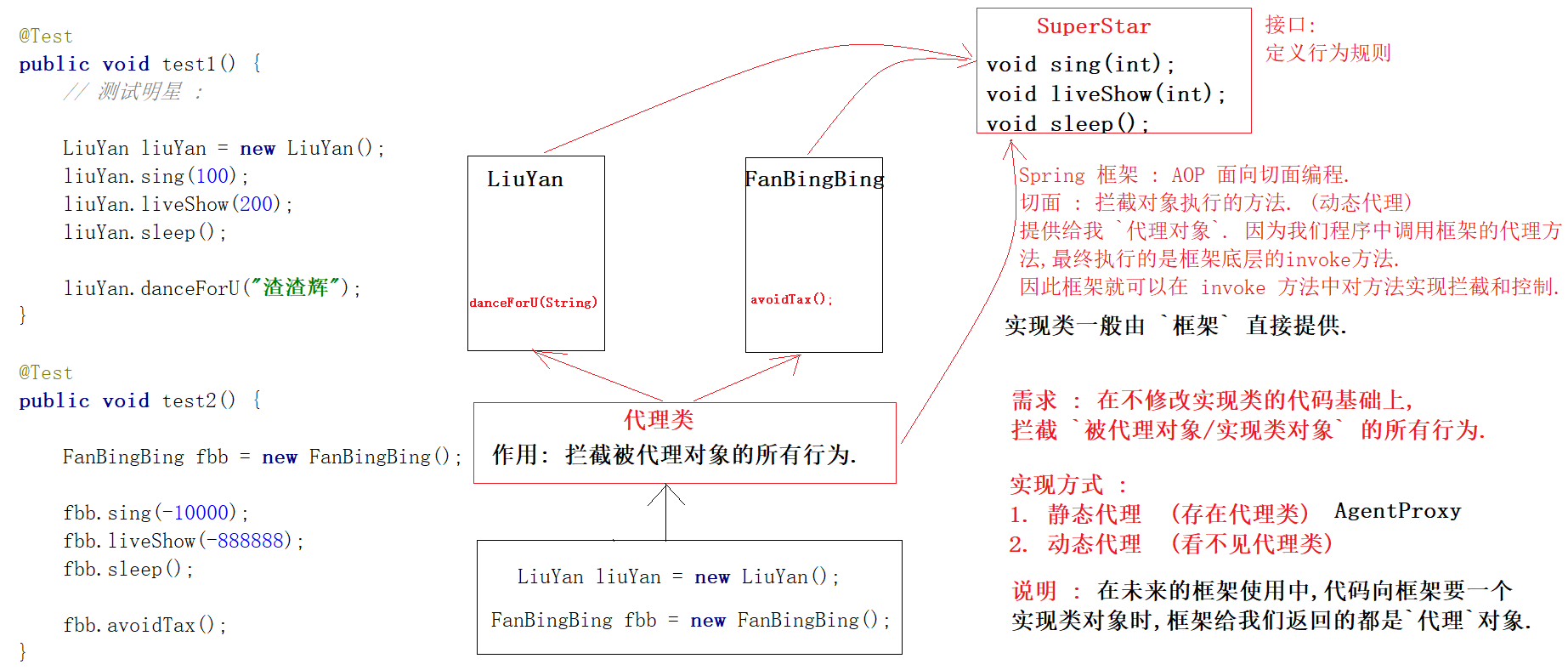
中介, 经纪人, 媒婆 …

作用 : 拦截并控制 `被代理对象` 的所有行为.

代理必须有一个提前条件 : 行为规范 (接口) 代理对象的行为规则必须要与 `被代理对象` 的行为规范保持一致.

SuperStar 明星接口 : sing(int money); liveShow(int money); sleep(); …

该接口的实现类对象 : LiuYan, FanBingBing … 需要实现 SuperStar 接口, 就可以拥有共同的行为规则.



**SuperStar 接口定义 :**

*// 明星的行为规范接口 :*public interface SuperStar {  
  
 void sing(int money);  
  
 void liveShow(int money);  
  
 void sleep();  
}

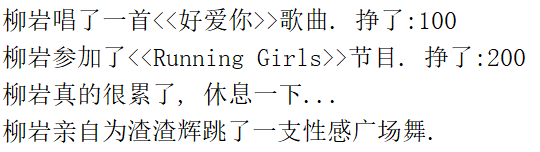
**LiuYan, FanBingBing 实现类定义 :**

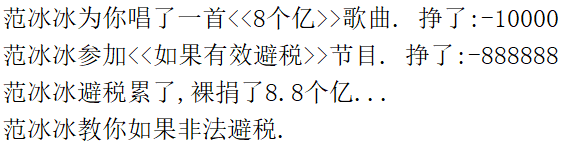
public class LiuYan implements SuperStar {  
  
 @Override  
 public void sing(int money) {  
 System.*out*.println("柳岩唱了一首<<好爱你>>歌曲. 挣了:" + money);  
 }  
  
 @Override  
 public void liveShow(int money) {  
 System.*out*.println("柳岩参加了<<Running Girls>>节目. 挣了:" + money);  
 }  
  
 @Override  
 public void sleep() {  
 System.*out*.println("柳岩真的很累了, 休息一下...");  
 }  
  
 public void danceForU(String u) {  
 System.*out*.println("柳岩亲自为"+u+"跳了一支性感广场舞.");  
 }  
}

public class FanBingBing implements SuperStar {  
  
 @Override  
 public void sing(int money) {  
 System.*out*.println("范冰冰为你唱了一首<<8个亿>>歌曲. 挣了:" + money);  
 }  
  
 @Override  
 public void liveShow(int money) {  
 System.*out*.println("范冰冰参加<<如果有效避税>>节目. 挣了:" + money);  
 }  
  
 @Override  
 public void sleep() {  
 System.*out*.println("范冰冰避税累了,裸捐了8.8个亿...");  
 }  
  
 public void avoidTax() {  
 System.*out*.println("范冰冰教你如果非法避税.");  
 }  
}

**测试类 :**

@Test  
public void test1() {  
 *// 测试明星 :* LiuYan liuYan = new LiuYan();  
 liuYan.sing(100);  
 liuYan.liveShow(200);  
 liuYan.sleep();  
  
 liuYan.danceForU("渣渣辉");  
}  
  
@Test  
public void test2() {  
  
 FanBingBing fbb = new FanBingBing();  
  
 fbb.sing(-10000);  
 fbb.liveShow(-888888);  
 fbb.sleep();  
  
 fbb.avoidTax();  
}





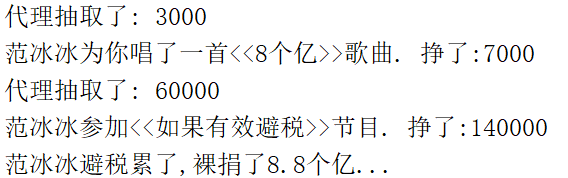
## 1 静态代理 :

**AgentProxy 静态代理类 :**

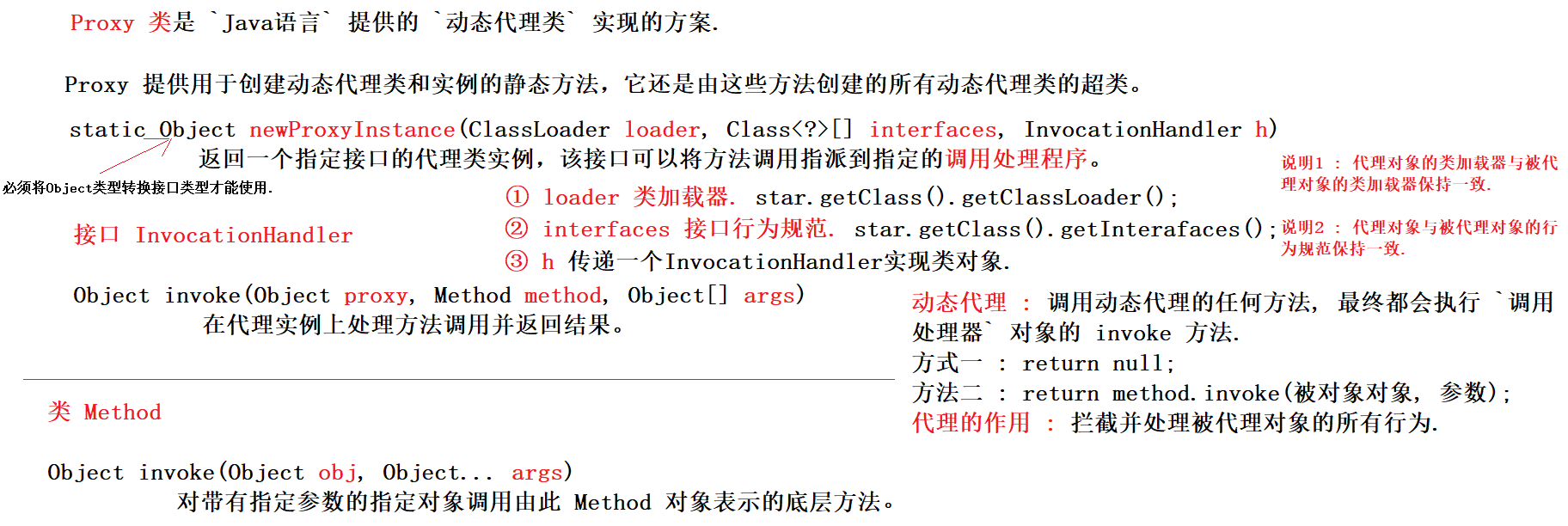
public class AgentProxy implements SuperStar {  
 *// 属性* private SuperStar star;  
  
 *// 如何表示 AgentProxy 代理了一个 SuperStar 实现类对象呢 ???* public AgentProxy(SuperStar star) {  
 this.star = star;  
 }  
  
 @Override  
 public void sing(int money) {  
 *// 1. 判断* if (money < 10000) {  
 System.*out*.println("滚,你个穷屌丝,回来撸代码去...");  
 *// 直接返回, 不再执行后续行为* return;  
 }  
 *// 2. 参数满足条件, 先执行内部的一些行为* int huikou = (int) (money \* 0.3);  
 System.*out*.println("代理抽取了: " + huikou);  
 *// 3. 让 `被代理对象` 来执行真实行为* star.sing(money - huikou);  
 }  
  
 @Override  
 public void liveShow(int money) {  
 *// 1. 判断* if (money < 100000) {  
 System.*out*.println("滚,又是一个穷屌丝,回来继续撸代码...");  
 *// 直接返回, 不再执行后续行为* return;  
 }  
 *// 2. 参数满足条件, 先执行内部的一些行为* int huikou = (int) (money \* 0.3);  
 System.*out*.println("代理抽取了: " + huikou);  
 *// 3. 让 `被代理对象` 来执行真实行为* star.liveShow(money - huikou);  
 }  
  
 @Override  
 public void sleep() {  
 *// 直接放行...* star.sleep();  
 }  
}

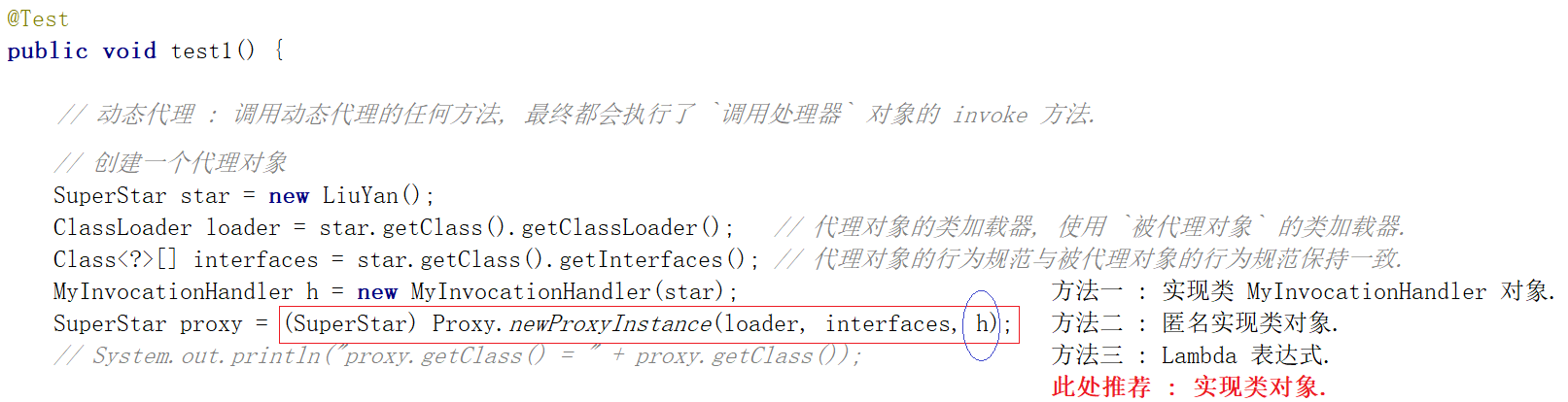
**测试类 :**

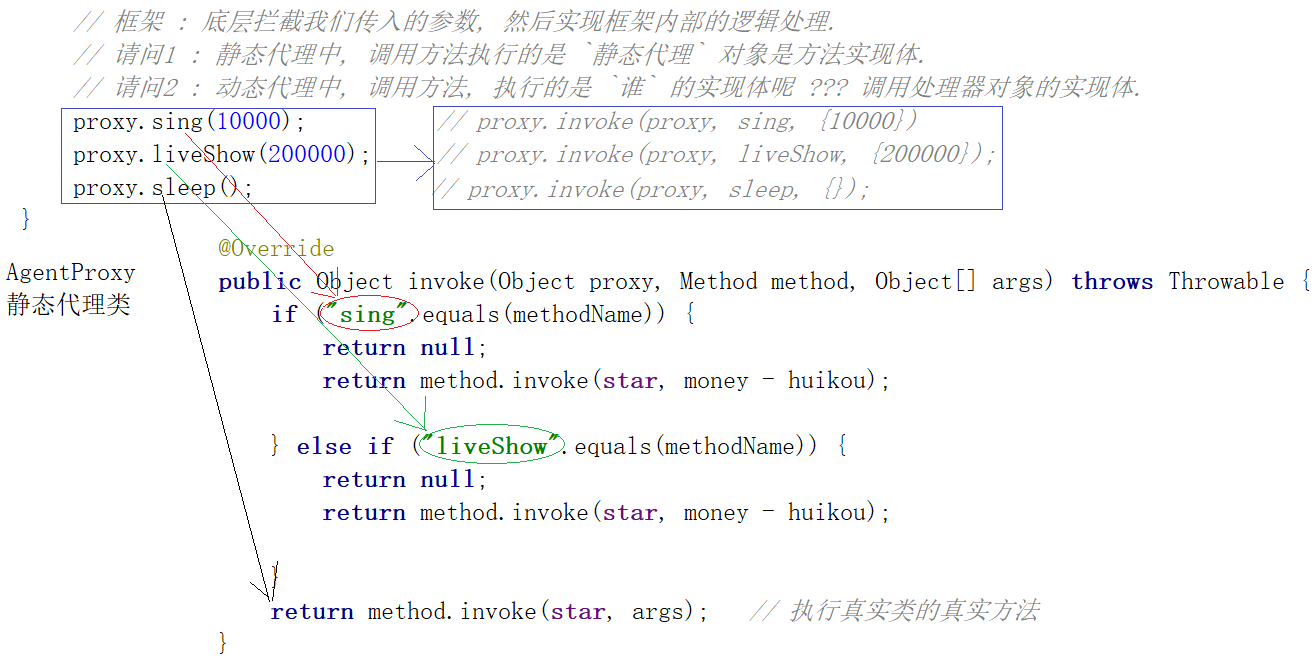
@Test  
public void test1() {  
  
 *// AgentProxy proxy = new AgentProxy(new LiuYan());* AgentProxy proxy = new AgentProxy(new FanBingBing());  
  
 *// 框架 : 底层拦截我们传入的参数, 然后实现框架内部的逻辑处理.* proxy.sing(10000);  
 proxy.liveShow(200000);  
 proxy.sleep();  
}



## 2 动态代理 :





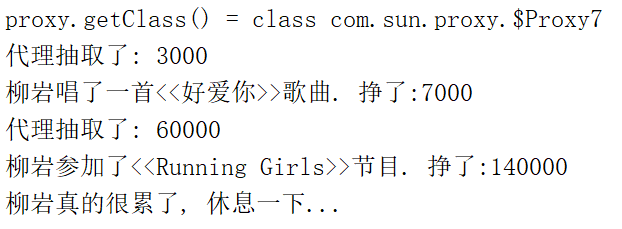


@Test  
public void test1() {  
  
 *// 动态代理 : 调用动态代理的任何方法, 最终都会执行了 `调用处理器` 对象的 invoke 方法.  
  
 // 创建一个代理对象* SuperStar star = new LiuYan();  
 ClassLoader loader = star.getClass().getClassLoader(); *// 代理对象的类加载器, 使用 `被代理对象` 的类加载器.* Class<?>[] interfaces = star.getClass().getInterfaces(); *// 代理对象的行为规范与被代理对象的行为规范保持一致.* MyInvocationHandler h = new MyInvocationHandler(star);  
 SuperStar proxy = (SuperStar) Proxy.*newProxyInstance*(loader, interfaces, h);System.*out*.println("proxy.getClass() = " + proxy.getClass()); *// 框架 : 底层拦截我们传入的参数, 然后实现框架内部的逻辑处理.  
 // 请问1 : 静态代理中, 调用方法执行的是 `静态代理` 对象是方法实现体.  
 // 请问2 : 动态代理中, 调用方法, 执行的是 `谁` 的实现体呢 ??? 调用处理器对象的实现体.* proxy.sing(10000); *// proxy.invoke(proxy, sing, {10000})* proxy.liveShow(200000); *// proxy.invoke(proxy, liveShow, {200000});* proxy.sleep(); *// proxy.invoke(proxy, sleep, {});*}

**调用处理器类 :**

import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
  
public class MyInvocationHandler implements InvocationHandler {  
 *// 属性* private SuperStar star;  
  
 public MyInvocationHandler(SuperStar star) {  
 this.star = star;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 *// System.out.println("method = " + method.getName());  
 // System.out.println("args = " + Arrays.toString(args));*

*// System.out.println("proxy.getClass() = " + proxy.getClass());*  
 *// 1. 获取调用的方法名称* String methodName = method.getName();  
 *// 2. 判断* if ("sing".equals(methodName)) {  
 *// System.out.println("调用了 sing 方法 ...");* int money = (int) args[0];  
 *// 1. 判断* if (money < 10000) {  
 System.*out*.println("滚,你个穷屌丝,回来撸代码去...");  
 *// 直接返回, 不再执行后续行为* return null;  
 }  
 *// 2. 参数满足条件, 先执行内部的一些行为* int huikou = (int) (money \* 0.3);  
 System.*out*.println("代理抽取了: " + huikou);  
 *// 3. 让 `被代理对象` 来执行真实行为* return method.invoke(star, money - huikou);  
  
 } else if ("liveShow".equals(methodName)) {  
 *// System.out.println("调用了 liveShow 方法 ...");* int money = (int) args[0];  
 *// 1. 判断* if (money < 100000) {  
 System.*out*.println("滚,又是一个穷屌丝,回来继续撸代码...");  
 *// 直接返回, 不再执行后续行为* return null;  
 }  
 *// 2. 参数满足条件, 先执行内部的一些行为* int huikou = (int) (money \* 0.3);  
 System.*out*.println("代理抽取了: " + huikou);  
 *// 3. 让 `被代理对象` 来执行真实行为* return method.invoke(star, money - huikou);  
  
 }*/\* else {  
 System.out.println("调用了其它方法 ...");  
 }\*/  
  
 // return null; // 不执行真实类的方法 ...* return method.invoke(star, args); *// 执行真实类的真实方法* }  
}

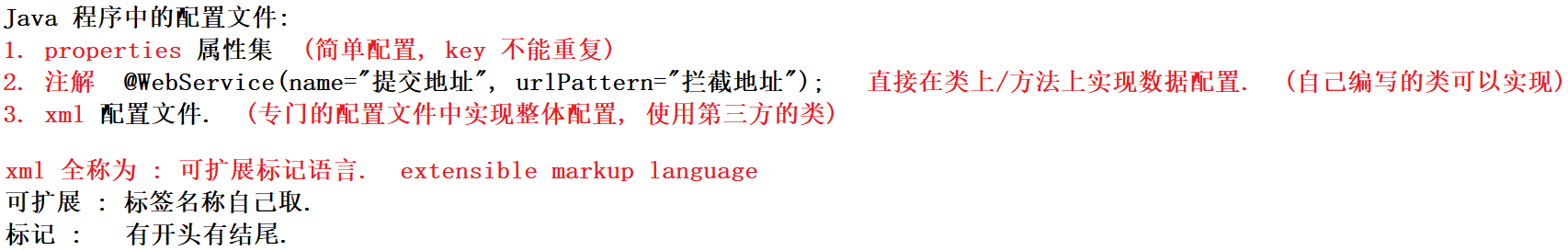


**$ 表示类的所属关系**

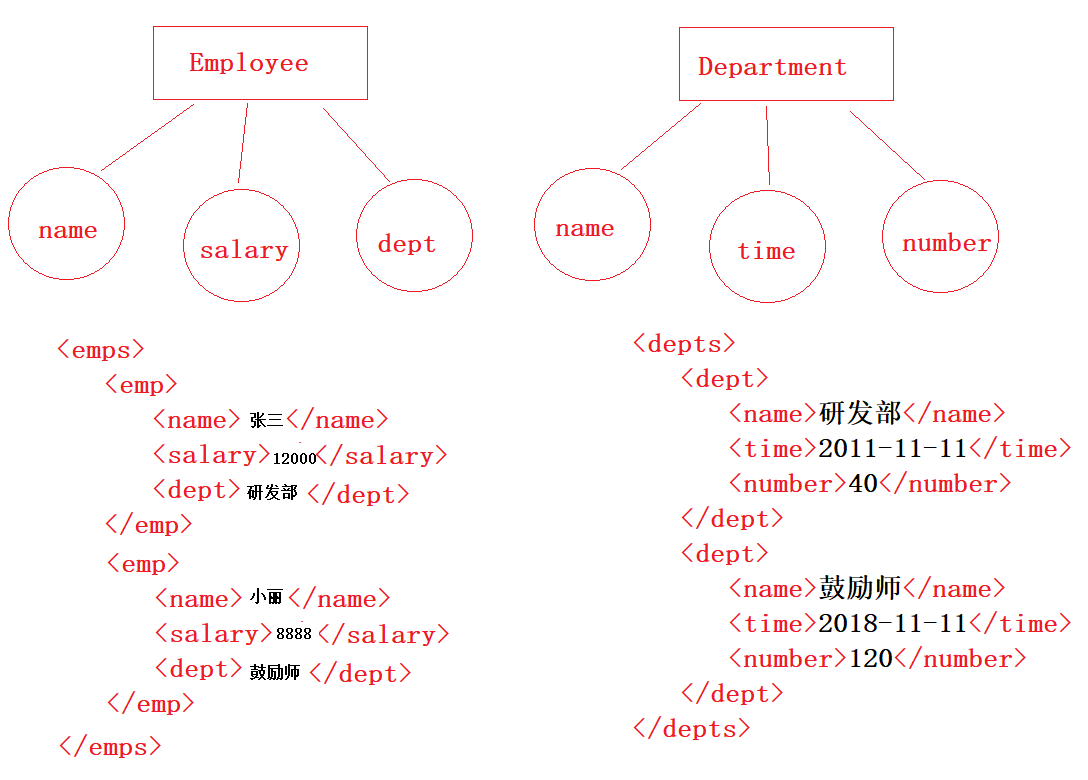


# 14. XML文件 : (了解)

XML 的使用场景 : 充当Java程序的 `配置文件`.







## 1 书写一个 xml 配置文件 :

*<?*xml version="1.0" encoding="UTF-8" *?>  
  
<!-- 一个合法的xml文件,应该有一个根标签 -->*<store>  
 *<!--xml 存储信息有两种方式, 一种是标签头的属性存储, 格式: key=value -->* <product category="手机数码">  
 *<!--另一种存储信息方式为标签体存储-->* <pname>华为P20</pname>  
 <price>3500</price>  
 <color>宝蓝色</color>  
  
 *<!--如果标题体中没有数据信息, 信息存储在标签属性中, 那么该标签可以书写为 `自关闭` 标签-->* <property name="username" value="张三"></property>  
 <property name="password" value="123456" />  
  
 </product>  
 <product category="电脑办公">  
 <pname>神州笔记本</pname>  
 <price>7500</price>  
 <color>黑色</color>  
 </product>  
 <product category="大型家电">  
 <pname>海尔兄弟洗衣机</pname>  
 <price>998</price>  
 <color>白色</color>  
 </product>  
</store>

## 2 约束介绍 (DTD, Schema) : 了解

**DTD 约束 : Document Type Defination**

<!DOCTYPE books[  
 <!ELEMENT books (book+)>  
 <!ELEMENT book (name, price)>  
 <!ELEMENT name (#PCDATA)>  
 <!ELEMENT price (#PCDATA)>  
 <!ATTLIST book author CDATA #REQUIRED> <!-- #IMPLIED 可选 -->  
 ]*>*

**xml 文件编写 : 拷贝 DTD 约束到xml文件中, 然后根据提示书写 xml 文件即可.**

*<?*xml version="1.0" encoding="UTF-8" *?>*<!DOCTYPE books[  
 <!ELEMENT books (book+)>  
 <!ELEMENT book (name, price)>  
 <!ELEMENT name (#PCDATA)>  
 <!ELEMENT price (#PCDATA)>  
 <!ATTLIST book author CDATA #REQUIRED> <!-- #IMPLIED 可选 -->  
 ]*>*<books>  
 <book author="张三丰">  
 <name>Java从入门到入土</name>  
 <price>98</price>  
 </book>  
 <book author="灭绝师太">  
 <name>Thinking in Java.</name>  
 <price>998</price>  
 </book>  
</books>

**Schema 约束 : XML Scheman**

xml schema defination xsd文件.

<!--  
 传智播客 Schema 教学实例文档 :  
 模拟 spring 框架规范 : 如果开发人员需要在 xml 使用当前 schema 约束, 必须包含指定命名空间.  
 格式如下 :  
 <beans xmlns="http://www.example.org/bean-schema"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.example.org/bean-schema bean-schema.xsd">   
 </beans>  
 -->  
  
<schema xmlns="http://www.w3.org/2001/XMLSchema"  
 targetNamespace="http://www.example.org/bean-schema"  
 xmlns:tns="http://www.example.org/bean-schema"  
 elementFormDefault="qualified">  
 <!-- 声明根标签 -->  
 <element name="beans">  
 <complexType>  
 <sequence>  
 <element name="bean" minOccurs="0" maxOccurs="unbounded">  
 <complexType>  
 <sequence>  
 <element name="property" minOccurs="0" maxOccurs="unbounded">  
 <complexType>  
 <attribute name="name" type="string" use="required"></attribute>  
 <attribute name="value" type="string" use="required"></attribute>  
 </complexType>  
 </element>  
 </sequence>  
 <attribute name="id" type="string" use="required"></attribute>  
 <attribute name="className" type="string" use="required"></attribute>  
 </complexType>  
 </element>  
 </sequence>  
 </complexType>  
 </element>  
</schema>

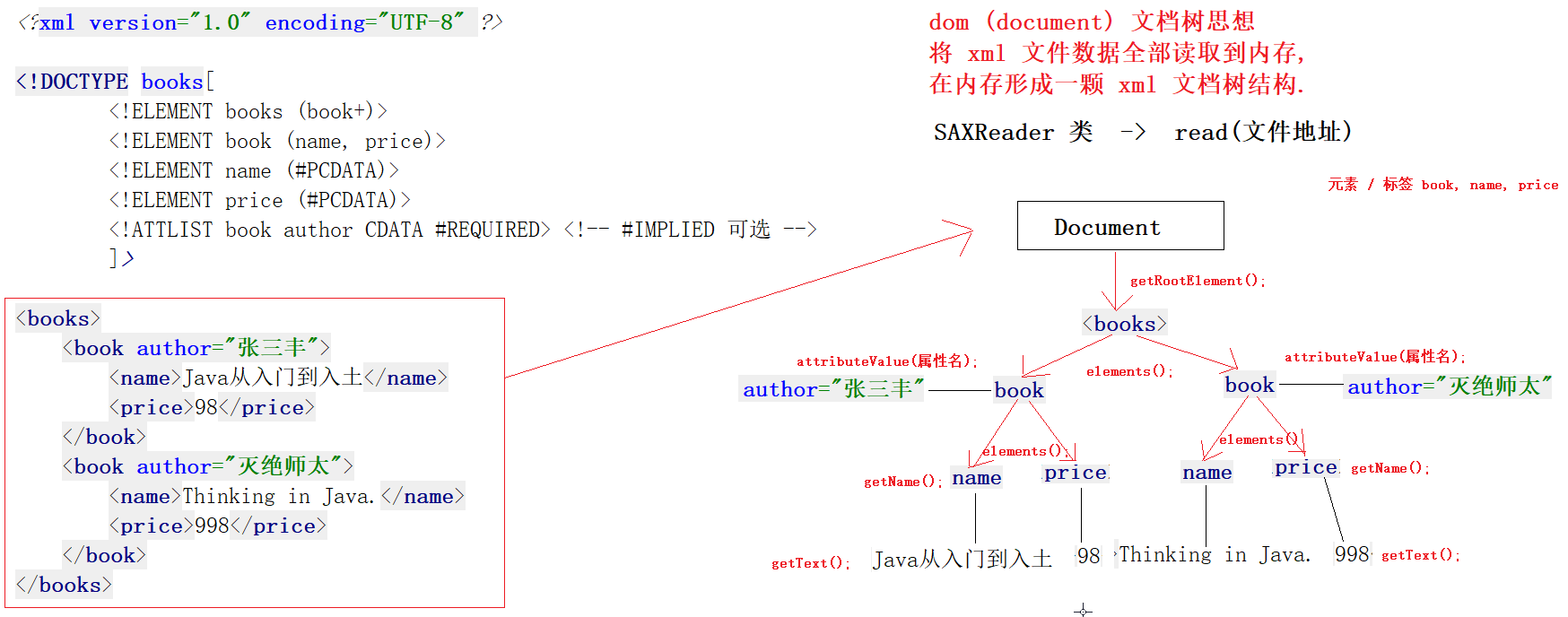
**xml 文件编写 : 拷贝xsd文件提供的xml格式,在注释中. 然后根据提示书写即可.**

*<?*xml version="1.0" encoding="UTF-8" *?>*<beans xmlns="http://www.example.org/bean-schema"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.example.org/bean-schema bean-schema.xsd">  
  
 <bean id="itcast\_001" className="cn.itcast.bean.User">  
 <property name="username" value="zhangsan"></property>  
 <property name="password" value="123456" />  
 </bean>  
 <bean id="itcast\_002" className="cn.itcast.bean.Student">  
 <property name="name" value="小丽"></property>  
 <property name="age" value="18"></property>  
 <property name="gender" value="女"></property>  
 </bean>

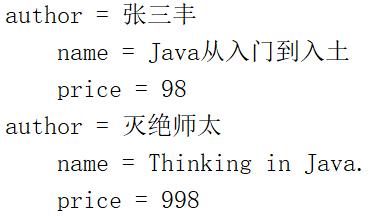
</beans>

## 3 xml解析 : 了解

**Dom4j (思想: 文档树思想)** 将 xml 文件读取到内存, 形成一个文档树, 然后进行解析.



import org.dom4j.Document;  
import org.dom4j.DocumentException;  
import org.dom4j.Element;  
import org.dom4j.io.SAXReader;  
  
import java.util.List;  
  
public class Test1 {  
 public static void main(String[] args) throws DocumentException {  
  
 *// 1. 创建一个 xml 解析对象 (SAXReader类)* SAXReader saxReader = new SAXReader();  
  
 *// 2. 根据 saxReader 对象调用 read 方法, 将 xml 文件读取到内存形成一颗文档树* Document document = saxReader.read("books.xml");  
  
 *// 3. 获取根标签* Element root = document.getRootElement();  
  
 *// 4. 根据 root 根标签对象, 获取子标签数组 (集合就是可变数组)* List<Element> bookElements = root.elements();  
  
 *// 5. 遍历集合* for (Element book : bookElements) {  
 *// attribute, field, property 属性 am is are* String author = book.attributeValue("author");  
 System.*out*.println("author = " + author);  
  
 *// 再次获取 book 的子标签* List<Element> elements = book.elements();  
 *// 遍历* for (Element element : elements) {  
 String name = element.getName();  
 String text = element.getText();  
 System.*out*.println("\t" + name + " = " + text);  
 }  
 }  
 }  
}



**今日小结 :**

1. 能够完成注解的MyTest案例

实现 : 解析注解 (AnnotatedElement) + 反射 (method执行invoke方法)

2. 能够说出XML的作用

作用 : 充当 Java程序的配置文件.

3. 了解XML的组成元素

xml 文档声明 : <? Version 版本 encoding 编码>

标签 : 成对出现, 可以自定义名称

属性 : key=value 存储数据

注释 : <!-- -->

1. 能够说出解析XML文档DOM方式原理

思想 : 将xml文件整体读取到内存, 形成一个文档树, 根据文档树结构调用对应的 api 获取数据.

1. 能够说出有哪些XML约束技术

技术 : DTD, Schema 约束文件是 `框架` 定义的, 我们编写 xml 要根据约束编写. IDEA 提供智能提示.

1. 动态代理 :

Proxy.newProxyInstance(类加载器, 接口规范, 调用处理器对象);

类加载器 : 加载动态代理类到内存.

接口规范 : 代理对象与被代理对象具有相同的行为规范.

调用处理器 : 调用动态代理的任何方法, 最终都指派给了调用处理器对象的 invoke 方法处理.

# 15. 正则表达式 : (重点)

Regex -> Regular Expression 正则表达式.

字符串的**特殊规则.** 虚拟机会将正则表达式字符串解析为对应的 Java 代码.

String 类的 matches 匹配.

String 类的 split 切割.

String 类的 replaceAll 替换.

## 1 正则引入 : 验证QQ号码

需求 : 验证QQ号码是否合法.

**分析 :**

1. 第一位不能是 0

2. QQ号码在 5 ~ 12 位之间 (包含)

3. QQ号码都是由数字组成.

**实现方式一 : 遍历+判断**

*// 方式一 :*public static boolean verifyQQNumber1(String qq) {  
 *// 1. 第一位不能是 0  
 // if (qq.startsWith("0"))* if (qq.charAt(0) == '0') {  
 return false;  
 } else if (qq.length() < 5 || qq.length() > 12) {  
 *// 2. QQ号码在 5 ~ 12 位之间 (包含)* return false;  
 } else {  
 *// 说明 : 程序如果能够来到这里, 说明 qq 号码第一位不是 0, 并且qq号码的长度在 5~12 位之间  
 // 3. QQ号码都是由数字组成.  
 // 3.1 遍历* for (int i = 0; i < qq.length(); i++) {  
 *// 3.2 取出每一个字符* char ch = qq.charAt(i);  
 *// 3.3 判断字符* if (ch < '0' || ch > '9') {  
 return false;  
 }  
 }  
 *// 3.4 返回 true* return true;  
 }  
}

**实现方式二 : 判断 + 异常捕获**

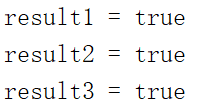
*// 方式二 : 异常处理*public static boolean verifyQQNumber2(String qq) {  
 *// 1. 第一位不能是 0  
 // if (qq.startsWith("0"))* if (qq.charAt(0) == '0') {  
 return false;  
 } else if (qq.length() < 5 || qq.length() > 12) {  
 *// 2. QQ号码在 5 ~ 12 位之间 (包含)* return false;  
 } else {  
 *// 说明 : 程序如果能够来到这里, 说明 qq 号码第一位不是 0, 并且qq号码的长度在 5~12 位之间  
 // 3. QQ号码都是由数字组成. (尝试: 将字符串解析为数字) Integer.parseInt(); Long.parseLong();* try {  
 Long.*parseLong*(qq);  
 } catch (NumberFormatException e) {  
 return false; *// 处理结果* }  
 return true;  
 }  
}

**正则实现 :**

*// 方式三 :*public static boolean verifyQQNumber3(String qq) {  
 *// return qq.matches("[1-9][0-9]{4,11}");* return qq.matches("[1-9]\\d{4,11}");  
}

**备份代码 :**

public static void main(String[] args) {  
  
 /\*  
 需求 : 验证QQ号码是否合法.  
  
 分析 :  
 1. 第一位不能是 0  
 2. QQ号码在 5 ~ 12 位之间 (包含)  
 3. QQ号码都是由数字组成.  
  
 \*/  
  
 String qq = "123456"; // 文本框中输入的数据都是 String 类型的.  
  
 boolean result1 = verifyQQNumber1(qq);  
 System.out.println("result1 = " + result1);  
  
 boolean result2 = verifyQQNumber2(qq);  
 System.out.println("result2 = " + result2);  
  
 boolean result3 = verifyQQNumber3(qq);  
 System.out.println("result3 = " + result3);  
}



## 2 正则的符号介绍 :

正则就是 `特殊规则` 的字符串, 虚拟机会将正则翻译成Java代码. Pattern 模式/正则对象

一 : 小括号 : 当规则需要复用是, 可以使用小括号给正则的规则实现分组. 1. [\\编号](file:///\\编号) 2. $编号 replaceAll方法中的第二个参数replacement中使用

二 : 中括号 : 表示数据的范围.

[0-9] [\\d](file:///\\d)

[0-9a-zA-Z\_] \\w

[1-9] [34578]

三 : 大括号 : 表示前面规则允许出现的次数.

{n} 固定次数.

{1,} 一次或更多. +

{0,1} 零次或一次. ?

{0,} 零次或更多. \*

## 3 匹配 – matches (手机号码)

手机号码规则 :

1. 长度必须是11位

2. 第一位只能是数字1

3. 第二位可以是3, 4, 5, 7, 8

## 4 切割 – split (字符串)

String str = "boxing#######basketball#####football###ILOVEYOU###爱我中华";  
String str = "boxing123basketball4567football888ILOVEYOU9876爱我中华";  
String str = "boxingaaaaabasketballYYYYYfootball888ILOVEYOU9999爱我中华";

## 5 替换 – replaceAll (隐藏手机号码)

13366285946 -> 133\*\*\*\*5946