文档（三）

流和日期

日期转日期

```
public class TestDate {

	public static void main(String[] args) {

		//y 代表年
		//M 代表月
		//d 代表日
		//H 代表 24 进制的小时
		//h 代表 12 进制的小时
		//m 代表分钟
		//s 代表秒
		//S 代表毫秒
		SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss SSS" );
		Date d= new Date();
		String str = sdf.format(d);
		System.out.println("当前时间通过 yyyy-MM-dd HH:mm:ss
SSS 格式化后的输出: "+str);

		SimpleDateFormat sdf1 =new SimpleDateFormat("yyyy-MM-
dd" );
		Date d1= new Date();
		String str1 = sdf1.format(d1);
		System.out.println("当前时间通过 yyyy-MM-dd 格式化后
的输出: "+str1);

	}
}
```

字符串转日期

```
public class TestDate {
```

```
        public static void main(String[] args) {
                SimpleDateFormat sdf =new
SimpleDateFormat("yyyy/MM/dd HH:mm:ss" );

                String str = "2016/1/5 12:12:12";

                try {
                        Date d = sdf.parse(str);
                        System.out.printf("字符串 %s 通过格
式   yyyy/MM/dd HH:mm:ss %n 转换为日期对象: %s",str,d.toString());
                } catch (ParseException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

创建文件对象

```
public class TestFile {

        public static void main(String[] args) {
                // 绝对路径
                File f1 = new File("d:/LOLFolder");
                System.out.println("f1 的绝对路径：" +
f1.getAbsolutePath());
                // 相对路径,相对于工作目录，如果在 eclipse 中，就是项
目目录
                File f2 = new File("LOL.exe");
                System.out.println("f2 的绝对路径：" +
f2.getAbsolutePath());

                // 把 f1 作为父目录创建文件对象
                File f3 = new File(f1, "LOL.exe");

                System.out.println("f3 的绝对路径：" +
f3.getAbsolutePath());
        }
}
```

文件常用方法

```
public class TestFile {

	public static void main(String[] args) {

		File f = new File("d:/LOLFolder/LOL.exe");
		System.out.println("当前文件是：" +f);
		//文件是否存在
		System.out.println("判断是否存在："+f.exists());

		//是否是文件夹
		System.out.println("判断是否是文件夹：
"+f.isDirectory());

		//是否是文件（非文件夹）
		System.out.println("判断是否是文件："+f.isFile());

		//文件长度
		System.out.println("获取文件的长度："+f.length());

		//文件最后修改时间
		long time = f.lastModified();
		Date d = new Date(time);
		System.out.println("获取文件的最后修改时间："+d);
		//设置文件修改时间为 1970.1.1 08:00:00
		f.setLastModified(0);

		//文件重命名
		File f2 =new File("d:/LOLFolder/DOTA.exe");
		f.renameTo(f2);
		System.out.println("把 LOL.exe 改名成了 DOTA.exe");

		System.out.println("注意： 需要在 D:\\LOLFolder 确实存
在一个 LOL.exe,\r\n 才可以看到对应的文件长度、修改时间等信息");
	}
}

public class TestFile {

	public static void main(String[] args) throws IOException {

		File f = new File("d:/LOLFolder/skin/garen.ski");
```

```
                // 以字符串数组的形式，返回当前文件夹下的所有文件（不包含子文件及子文件夹）
                f.list();

                // 以文件数组的形式，返回当前文件夹下的所有文件（不包含子文件及子文件夹）
                File[]fs= f.listFiles();

                // 以字符串形式返回获取所在文件夹
                f.getParent();

                // 以文件形式返回获取所在文件夹
                f.getParentFile();
                // 创建文件夹，如果父文件夹 skin 不存在，创建就无效
                f.mkdir();

                // 创建文件夹，如果父文件夹 skin 不存在，就会创建父文件夹
                f.mkdirs();

                // 创建一个空文件,如果父文件夹 skin 不存在，就会抛出异常
                f.createNewFile();
                // 所以创建一个空文件之前，通常都会创建父目录
                f.getParentFile().mkdirs();

                // 列出所有的盘符 c: d: e: 等等
                f.listRoots();

                // 删除文件
                f.delete();

                // JVM 结束的时候，删除文件，常用于临时文件的删除
                f.deleteOnExit();

        }
}
```

遍历文件夹

```
public class TestFile {

        public static void main(String[] args) {
```

```
                File f = new File("c:\\windows");
                File[] fs = f.listFiles();
                if(null==fs)
                        return;
                long minSize = Integer.MAX_VALUE;
                long maxSize = 0;
                File minFile = null;
                File maxFile = null;
                for (File file : fs) {
                        if(file.isDirectory())
                                continue;
                        if(file.length()>maxSize){
                                maxSize = file.length();
                                maxFile = file;
                        }
                        if(file.length()!=0 &&
file.length()<minSize){
                                minSize = file.length();
                                minFile = file;
                        }
                }
                System.out.printf("最大的文件是%s，其大小是%,d字
节%n",maxFile.getAbsoluteFile(),maxFile.length());
                System.out.printf("最小的文件是%s，其大小是%,d字
节%n",minFile.getAbsoluteFile(),minFile.length());

        }
}
```

遍历子文件夹

```
public class TestFile {

        static long minSize = Integer.MAX_VALUE;
        static long maxSize = 0;
        static File minFile = null;
        static File maxFile = null;

        //使用递归来遍历一个文件夹的子文件
        public static void listFiles(File file){
                if(file.isFile()){
                        if(file.length()>maxSize){
                                maxSize = file.length();
                                maxFile = file;
```

```
                        }
                        if(file.length()!=0 &&
file.length()<minSize){
                                minSize = file.length();
                                minFile = file;
                        }
                        return;
                }

                if(file.isDirectory()){
                        File[] fs = file.listFiles();
                        if(null!=fs)
                        for (File f : fs) {
                                listFiles(f);
                        }
                }
        }

        public static void main(String[] args) {
                File f = new File("c:\\windows");
                listFiles(f);
                System.out.printf("最大的文件是%s，其大小是%,d字
节%n",maxFile.getAbsoluteFile(),maxFile.length());
                System.out.printf("最小的文件是%s，其大小是%,d字
节%n",minFile.getAbsoluteFile(),minFile.length());

        }
}
```

FileOutputStream

```
public static void main(String[] args) {
                try {
                        File f = new File("d:/lol.txt");
                        // 创建基于文件的输出流
                        FileOutputStream fos = new
FileOutputStream(f);
                        // 通过这个输出流，就可以把数据从内存，输出到
硬盘的文件上

                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
```

```
        }
}
```

字节流

读取：

```
public class TestStream {

        public static void main(String[] args) {
                try {
                        //准备文件 lol.txt 其中的内容是 AB，对应的
ASCII 分别是 65 66
                        File f =new File("d:/lol.txt");
                        //创建基于文件的输入流
                        FileInputStream fis =new FileInputStream(f);
                        //创建字节数组，其长度就是文件的长度
                        byte[] all =new byte[(int) f.length()];
                        //以字节流的形式读取文件所有内容
                        fis.read(all);
                        for (byte b : all) {
                                //打印出来是 65 66
                                System.out.println(b);
                        }

                        //每次使用完流，都应该进行关闭
                        fis.close();

                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

写入

```
public class TestStream {

        public static void main(String[] args) {
                try {
                        // 准备文件 lol2.txt 其中的内容是空的
                        File f = new File("d:/lol2.txt");
```

```
                        // 准备长度是 2 的字节数组，用 88,89 初始化，其
对应的字符分别是 X,Y
                        byte data[] = { 88, 89 };

                        // 创建基于文件的输出流
                        FileOutputStream fos = new
FileOutputStream(f);
                        // 把数据写入到输出流
                        fos.write(data);
                        // 关闭输出流
                        fos.close();

                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

拆分文件

```
public class TestStream {

        public static void main(String[] args) {
                int eachSize = 100 * 1024; // 100k
                File srcFile = new File("d:/eclipse.exe");
                splitFile(srcFile, eachSize);
        }

        /**
          * 拆分的思路，先把源文件的所有内容读取到内存中，然后从内存
中挨个分到子文件里
          * @param srcFile 要拆分的源文件
          * @param eachSize 按照这个大小，拆分
          */
        private static void splitFile(File srcFile, int eachSize) {

                if (0 == srcFile.length())
                        throw new RuntimeException("文件长度为 0，不
可拆分");

                byte[] fileContent = new byte[(int)
srcFile.length()];
```

```
                // 先把文件读取到数组中
                try {
                        FileInputStream fis = new
FileInputStream(srcFile);
                        fis.read(fileContent);
                        fis.close();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                // 计算需要被划分成多少份子文件
                int fileNumber;
                // 文件是否能被整除得到的子文件个数是不一样的
                // (假设文件长度是 25，每份的大小是 5，那么就应该是 5
个)
                // (假设文件长度是 26，每份的大小是 5，那么就应该是 6
个)
                if (0 == fileContent.length % eachSize)
                        fileNumber = (int) (fileContent.length /
eachSize);
                else
                        fileNumber = (int) (fileContent.length /
eachSize) + 1;

                for (int i = 0; i < fileNumber; i++) {
                        String eachFileName = srcFile.getName() + "-"
+ i;
                        File eachFile = new File(srcFile.getParent(),
eachFileName);
                        byte[] eachContent;

                        // 从源文件的内容里，复制部分数据到子文件
                        // 除开最后一个文件，其他文件大小都是 100k
                        // 最后一个文件的大小是剩余的
                        if (i != fileNumber - 1) // 不是最后一个
                                eachContent =
Arrays.copyOfRange(fileContent, eachSize * i, eachSize * (i + 1));
                        else // 最后一个
                                eachContent =
Arrays.copyOfRange(fileContent, eachSize * i, fileContent.length);

                        try {
                                // 写出去
```

```
                                FileOutputStream fos = new
FileOutputStream(eachFile);
                                fos.write(eachContent);
                                // 记得关闭
                                fos.close();
                                System.out.printf("输出子文件%s，其大
小是 %d 字节%n", eachFile.getAbsoluteFile(), eachFile.length());
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        }
}
```

合并文件

```
public class TestStream {

        public static void main(String[] args) {
                murgeFile("d:/", "eclipse.exe");
        }

        /**
          * 合并的思路，就是从 eclipse.exe-0 开始，读取到一个文件，就
开始写出到 eclipse.exe 中，直到没有文件可以读
          * @param folder
          *                        需要合并的文件所处于的目录
          * @param fileName
          *                        需要合并的文件的名称
          * @throws FileNotFoundException
          */
        private static void murgeFile(String folder, String
fileName) {

                try {
                        // 合并的目标文件
                        File destFile = new File(folder, fileName);
                        FileOutputStream fos = new
FileOutputStream(destFile);
                        int index = 0;
                        while (true) {
                                //子文件
```

```
                                File eachFile = new File(folder,
fileName + "-" + index++);
                                //如果子文件不存在了就结束
                                if (!eachFile.exists())
                                        break;

                                //读取子文件的内容
                                FileInputStream fis = new
FileInputStream(eachFile);
                                byte[] eachContent = new byte[(int)
eachFile.length()];
                                fis.read(eachContent);
                                fis.close();

                                //把子文件的内容写出去
                                fos.write(eachContent);
                                fos.flush();
                                System.out.printf("把子文件 %s 写出到
目标文件中%n",eachFile);
                        }

                        fos.close();
                        System.out.printf("最后目标文件的大小：%,d 字
节" , destFile.length());
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

}
```

关闭流的方式

```
public static void main(String[] args) {
                File f = new File("d:/lol.txt");
                FileInputStream fis = null;
                try {
                        fis = new FileInputStream(f);
```

```
                byte[] all = new byte[(int) f.length()];
                fis.read(all);
                for (byte b : all) {
                        System.out.println(b);
                }

        } catch (IOException e) {
                e.printStackTrace();
        } finally {
                // 在 finally 里关闭流
                if (null != fis)
                        try {

                                fis.close();
                        } catch (IOException e) {
                                // TODO Auto-generated catch
block

                                e.printStackTrace();
                        }
        }
```

字符流

读取：

```
public class TestStream {

        public static void main(String[] args) {
                // 准备文件 lol.txt 其中的内容是 AB
                File f = new File("d:/lol.txt");
                // 创建基于文件的 Reader
                try (FileReader fr = new FileReader(f)) {
                        // 创建字符数组，其长度就是文件的长度
                        char[] all = new char[(int) f.length()];
                        // 以字符流的形式读取文件所有内容
                        fr.read(all);
                        for (char b : all) {
                                // 打印出来是 A B
                                System.out.println(b);
                        }
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
```

```
        }
}
```

写入

```
public class TestStream {

        public static void main(String[] args) {
                // 准备文件 lol2.txt
                File f = new File("d:/lol2.txt");
                // 创建基于文件的 Writer
                try (FileWriter fr = new FileWriter(f)) {
                        // 以字符流的形式把数据写入到文件中
                        String data="abcdefg1234567890";
                        char[] cs = data.toCharArray();
                        fr.write(cs);

                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

文件加密

```
public class TestStream {
        /**
          *
          * @param encodingFile
          *                        被加密的文件
          * @param encodedFile
          *                        加密后保存的位置
          */
        public static void encodeFile(File encodingFile, File
encodedFile) {

                try (FileReader fr = new FileReader(encodingFile);
FileWriter fw = new FileWriter(encodedFile)) {
                        // 读取源文件
                        char[] fileContent = new char[(int)
encodingFile.length()];
                        fr.read(fileContent);
                        System.out.println("加密前的内容：");
```

```
                System.out.println(new String(fileContent));

                // 进行加密
                encode(fileContent);
                // 把加密后的内容保存到目标文件
                System.out.println("加密后的内容：");
                System.out.println(new String(fileContent));

                fw.write(fileContent);
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        private static void encode(char[] fileContent) {
            for (int i = 0; i < fileContent.length; i++) {
                char c = fileContent[i];
                if (isLetterOrDigit(c)) {
                    switch (c) {
                    case '9':
                        c = '0';
                        break;
                    case 'z':
                        c = 'a';
                        break;
                    case 'Z':
                        c = 'A';
                        break;
                    default:
                        c++;
                        break;
                    }
                }
                fileContent[i] = c;
            }
        }

        public static boolean isLetterOrDigit(char c) {
            // 不使用 Character 类的 isLetterOrDigit 方法是因为，中文也会被判断为字母
            String letterOrDigital = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
```

```
                return -1 == letterOrDigital.indexOf(c) ? false :
true;
        }

        public static void main(String[] args) {
                File encodingFile = new
File("E:/project/j2se/src/Test1.txt");
                File encodedFile = new
File("E:/project/j2se/src/Test2.txt");
                encodeFile(encodingFile, encodedFile);
        }
}
```

文件解密

```
public class TestStream {

        /**
          *
          * @param decodingFile
          *                        被解密的文件
          * @param decodedFile
          *                        解密后保存的位置
          */
        public static void decodeFile(File decodingFile, File
decodedFile) {

                try (FileReader fr = new FileReader(decodingFile);
FileWriter fw = new FileWriter(decodedFile)) {
                        // 读取源文件
                        char[] fileContent = new char[(int)
decodingFile.length()];
                        fr.read(fileContent);
                        System.out.println("源文件的内容:");
                        System.out.println(new String(fileContent));
                        // 进行解密
                        decode(fileContent);
                        System.out.println("解密后的内容:");
                        System.out.println(new String(fileContent));
                        // 把解密后的内容保存到目标文件
                        fw.write(fileContent);
                } catch (IOException e) {
                        // TODO Auto-generated catch block
```

```
                e.printStackTrace();
            }
        }

        private static void decode(char[] fileContent) {
            for (int i = 0; i < fileContent.length; i++) {
                char c = fileContent[i];
                if (isLetterOrDigit(c)) {
                    switch (c) {
                    case '0':
                        c = '9';
                        break;
                    case 'a':
                        c = 'z';
                        break;
                    case 'A':
                        c = 'Z';
                        break;
                    default:
                        c--;
                        break;
                    }
                }
                fileContent[i] = c;
            }
        }

        public static boolean isLetterOrDigit(char c) {
            // 不使用Character类的isLetterOrDigit方法是因为，中文也会被判断为字母
            String letterOrDigital ="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
            return -1 == letterOrDigital.indexOf(c) ? false : true;
        }

        public static void main(String[] args) {
            File decodingFile = new File("E:/project/j2se/src/Test2.txt");
            File decodedFile = new File("E:/project/j2se/src/Test1.txt");

            decodeFile(decodingFile, decodedFile);
```

```
	}
}
```

缓存流读取

```
public class TestStream {

	public static void main(String[] args) {
		// 准备文件 lol.txt 其中的内容是
		// garen kill teemo
		// teemo revive after 1 minutes
		// teemo try to garen, but killed again
		File f = new File("d:/lol.txt");
		// 创建文件字符流
		// 缓存流必须建立在一个存在的流的基础上
		try (
				FileReader fr = new FileReader(f);
				BufferedReader br = new
BufferedReader(fr);
			)
		{
			while (true) {
				// 一次读一行
				String line = br.readLine();
				if (null == line)
					break;
				System.out.println(line);
			}
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}
}
```

缓存流写入

```
public class TestStream {

	public static void main(String[] args) {
		// 向文件 lol2.txt 中写入三行语句
		File f = new File("d:/lol2.txt");
```

```
                try (
                                // 创建文件字符流
                                FileWriter fw = new FileWriter(f);
                                // 缓存流必须建立在一个存在的流的基础
上
                                PrintWriter pw = new
PrintWriter(fw);
                ) {
                        pw.println("garen kill teemo");
                        pw.println("teemo revive after 1 minutes");
                        pw.println("teemo try to garen, but killed
again");
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}

public class TestStream {
        public static void main(String[] args) {
                //向文件 lol2.txt 中写入三行语句
                File f =new File("d:/lol2.txt");
                //创建文件字符流
                //缓存流必须建立在一个存在的流的基础上
                try(FileWriter fr = new FileWriter(f);PrintWriter pw
= new PrintWriter(fr);) {
                        pw.println("garen kill teemo");
                        //强制把缓存中的数据写入硬盘，无论缓存是否已
满
                                pw.flush();
                        pw.println("teemo revive after 1 minutes");
                                pw.flush();
                        pw.println("teemo try to garen, but killed
again");
                                pw.flush();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

移除注释

```
public class TestStream {

	public static void removeComments(File javaFile) {
		StringBuffer sb = new StringBuffer();
		//读取内容
		try (FileReader fr = new FileReader(javaFile);
BufferedReader br = new BufferedReader(fr);) {
			while (true) {
				String line = br.readLine();
				if (null == line)
					break;
				//如果不是以//开头，就保存在
StringBuffer 中
				if (!line.trim().startsWith("//"))
					sb.append(line).append("\r\n"
);
			}
		} catch (IOException e) {
			e.printStackTrace();
		}

		try (
			FileWriter fw = new FileWriter(javaFile);
			PrintWriter pw = new PrintWriter(fw);
		) {
			//写出内容
			pw.write(sb.toString());
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		File javaFile = new
File("E:\\project\\j2se\\src\\character\\MyStringBuffer2.java");
		System.out.println(javaFile.exists());
		System.out.println(javaFile.length());
		removeComments(javaFile);
	}
}
```

数据输入流

读取

```
public class TestStream {

	public static void main(String[] args) {
		write();
		read();
	}

	private static void read() {
		File f =new File("d:/lol.txt");
		try (
				FileInputStream fis   = new
FileInputStream(f);
				DataInputStream dis =new
DataInputStream(fis);
		){
			boolean b= dis.readBoolean();
			int i = dis.readInt();
			String str = dis.readUTF();

			System.out.println("读取到布尔值:"+b);
			System.out.println("读取到整数:"+i);
			System.out.println("读取到字符串:"+str);

		} catch (IOException e) {
			e.printStackTrace();
		}

	}

	private static void write() {
		File f =new File("d:/lol.txt");
		try (
				FileOutputStream fos   = new
FileOutputStream(f);
				DataOutputStream dos =new
DataOutputStream(fos);
		){
			dos.writeBoolean(true);
			dos.writeInt(300);
			dos.writeUTF("123 this is gareen");
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }
}
```

练习-向文件中写入两个数字，然后把这两个数字分别读取出来

```
public class TestStream {
        static File f =new File("d:/data.txt");
        static int x = 31;
        static int y = 15;
        public static void main(String[] args) {

                //缓存流方式
                method1();
                //数据流方式
                method2();
        }

        private static void method2() {
                try (
                                FileInputStream fis   = new
FileInputStream(f);
                                DataInputStream dis =new
DataInputStream(fis);
                                FileOutputStream fos   = new
FileOutputStream(f);
                                DataOutputStream dos =new
DataOutputStream(fos);

                ){
                        dos.writeInt(x);
                        dos.writeInt(y);

                        int x = dis.readInt();
                        int y = dis.readInt();
                        System.out.printf("使用数据流读取出的 x 是 %d
y 是 %d%n",x,y);

                } catch (IOException e) {
```

```
                        e.printStackTrace();
                }

        }

        private static void method1() {
                  try (
                                  FileWriter fw = new FileWriter(f);
                                  PrintWriter pw = new
PrintWriter(fw);
                                  FileReader fr = new FileReader(f);
                                  BufferedReader br = new
BufferedReader(fr);

                  ) {
                          pw.print(x+"@"+y);
                          pw.flush();
                          String str = br.readLine();
                          String[] ss =str.split("@");
                          int x = Integer.parseInt(ss[0]);
                          int y = Integer.parseInt(ss[1]);
                          System.out.printf("使用缓存流读取出的 x
是 %d y 是 %d%n",x,y);

                  } catch (IOException e) {
                          // TODO Auto-generated catch block
                          e.printStackTrace();
                  }

        }

        private static void read() {
                File f =new File("d:/data.txt");
                try (
                                  FileInputStream fis   = new
FileInputStream(f);
                                  DataInputStream dis =new
DataInputStream(fis);
                ){
                          boolean b= dis.readBoolean();
                          int i = dis.readInt();
                          String str = dis.readUTF();

                          System.out.println("读取到布尔值:"+b);
```

```
                System.out.println("读取到整数:"+i);
                System.out.println("读取到字符串:"+str);

            } catch (IOException e) {
                e.printStackTrace();
            }

        }

        private static void write() {
            File f =new File("d:/data.txt");
            try (
                    FileOutputStream fos   = new
FileOutputStream(f);
                    DataOutputStream dos =new
DataOutputStream(fos);
            ){
                dos.writeBoolean(true);
                dos.writeInt(300);
                dos.writeUTF("123 this is gareen");
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
}
```

复制文件

```
public class TestStream {
        /**
          *
          * @param srcPath 源文件
          * @param destPath 目标文件
          */
        public static void copyFile(String srcPath, String
destPath){

            File srcFile = new File(srcPath);
            File destFile = new File(destPath);

            //缓存区，一次性读取 1024 字节
            byte[] buffer = new byte[1024];
```

```
                try (
                                FileInputStream fis = new
FileInputStream(srcFile);
                                FileOutputStream fos = new
FileOutputStream(destFile);
                ){
                        while(true){
                                //实际读取的长度是 actuallyReaded,有
可能小于 1024
                                int actuallyReaded =
fis.read(buffer);
                                //-1 表示没有可读的内容了
                                if(-1==actuallyReaded)
                                        break;
                                fos.write(buffer, 0, actuallyReaded);
                                fos.flush();
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }

        /**
          *
          * @param srcPath 源文件夹
          * @param destPath 目标文件夹
          */
        public static void copyFolder(String srcPath, String
destPath){

        }

        public static void main(String[] args) {

                copyFile("d:/lol.txt", "d:/lol2.txt");

        }
}
```

复制文件夹

```
public class TestStream {
```

```
        /**
          *
          * @param srcPath 源文件
          * @param destPath 目标文件
          */
        public static void copyFile(String srcPath, String
destPath){

                File srcFile = new File(srcPath);
                File destFile = new File(destPath);

                //缓存区，一次性读取 1024 字节
                byte[] buffer = new byte[1024];

                try (
                                FileInputStream fis = new
FileInputStream(srcFile);
                                FileOutputStream fos = new
FileOutputStream(destFile);
                ){
                        while(true){
                                //实际读取的长度是 actuallyReaded,有
可能小于 1024
                                int actuallyReaded =
fis.read(buffer);
                                //-1 表示没有可读的内容了
                                if(-1==actuallyReaded)
                                        break;
                                fos.write(buffer, 0, actuallyReaded);
                                fos.flush();
                        }
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

        /**
          *
          * @param srcPath 源文件夹
```

```
         * @param destPath 目标文件夹
         */
        public static void copyFolder(String srcPath, String
destPath){

                File srcFolder = new File(srcPath);
                File destFolder = new File(destPath);
                //源文件夹不存在
                if(!srcFolder.exists())
                        return;
                //源文件夹不是一个文件夹
                if(!srcFolder.isDirectory())
                        return;
                //目标文件夹是一个文件
                if(destFolder.isFile())
                        return;
                //目标文件夹不存在
                if(!destFolder.exists())
                        destFolder.mkdirs();

                //遍历源文件夹
                File[] files=   srcFolder.listFiles();
                for (File srcFile : files) {
                        //如果是文件，就复制
                        if(srcFile.isFile()){
                                File newDestFile = new
File(destFolder,srcFile.getName());
                                copyFile(srcFile.getAbsolutePath(),
newDestFile.getAbsolutePath());
                        }
                        //如果是文件夹，就递归
                        if(srcFile.isDirectory()){
                                File newDestFolder = new
File(destFolder,srcFile.getName());
                                copyFolder(srcFile.getAbsolutePath(),
newDestFolder.getAbsolutePath());
                        }
                }
        }
        public static void main(String[] args) {
                copyFolder("d:/LOLFolder", "d:/LOLFolder2");
        }
}
```

查找文件内容

```
public class TestStream {

	/**
	 * @param file 查找的目录
	 * @param search 查找的字符串
	 */
	public static void search(File file, String search) {
		if (file.isFile()) {
			if(file.getName().toLowerCase().endsWith(".ja
va")){
				String fileContent =
readFileConent(file);
				if(fileContent.contains(search)){
					System.out.printf("找到子目标
字符串%s,在文件:%s%n",search,file);
				}
			}
		}
		if (file.isDirectory()) {
			File[] fs = file.listFiles();
			for (File f : fs) {
				search(f, search);
			}
		}
	}

	public static String readFileConent(File file){
		try (FileReader fr = new FileReader(file)) {
			char[] all = new char[(int) file.length()];
			fr.read(all);
			return new String(all);
		} catch (IOException e) {
			e.printStackTrace();
			return null;
		}

	}

	public static void main(String[] args) {
		File folder =new File("e:\\project");
		search(folder,"Magic");
```

```
        }
}
```

流关系

集合

ArrayList 和 LinkedList 的区别

ArrayList **插入，删除数据慢**
LinkedList， **插入，删除数据快**
ArrayList 是顺序结构，所以**定位很快**，指哪找哪。 就像电影院位置一样，有了电影票，一下就找到位置了。
LinkedList 是链表结构，就像手里的一串佛珠，要找出第 99 个佛珠，必须得一个一个的数过去，所以**定位慢**

HashMap 和 Hashtable 的区别

HashMap 和 Hashtable 都实现了 Map 接口，都是键值对保存数据的方式
区别 1：
HashMap 可以存放 null
Hashtable 不能存放 null
区别 2：
HashMap 不是线程安全的类

Hashtable 是线程安全的类

鉴于目前学习的进度，不对线程安全做展开，在线程章节会详细讲解

HashSet LinkedHashSet TreeSet

HashSet： 无序
LinkedHashSet： 按照插入顺序
TreeSet： 从小到大排序

List 查找的低效率

假设在 List 中存放着无重复名称，没有顺序的 2000000 个 Hero
要把名字叫做“hero 1000000”的对象找出来
List 的做法是对每一个进行挨个遍历，直到找到名字叫做“hero 1000000”的英雄。
最差的情况下，需要遍历和比较 2000000 次，才能找到对应的英雄。

HashMap 的性能表现

使用 HashMap 做同样的查找
1. 初始化 2000000 个对象到 HashMap 中。
2. 进行 10 次查询
3. 统计每一次的查询消耗的时间
可以观察到，几乎不花时间，花费的时间在 1 毫秒以内
在展开 HashMap 原理的讲解之前，首先回忆一下大家初中和高中使用的汉英字典。

比如要找一个单词对应的中文意思，假设单词是 Lengendary，首先在目录找到 Lengendary 在第 555 页。

然后，翻到第 555 页，这页不只一个单词，但是量已经很少了，逐一比较，很快就定位目标单词 Lengendary。

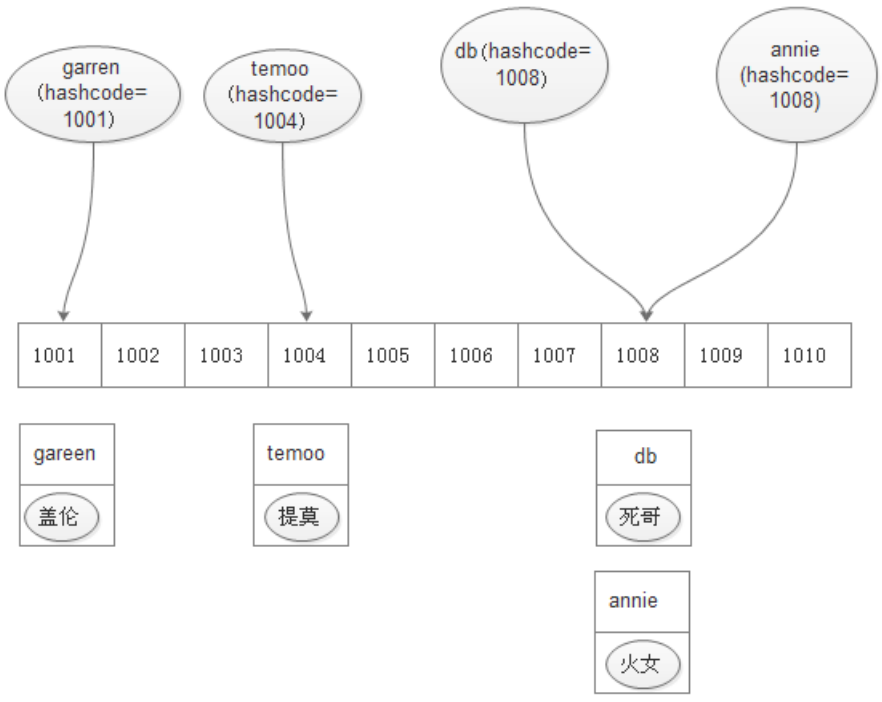
555 相当于就是 Lengendary 对应的 **hashcode**
-----hashcode 概念-----
所有的对象，都有一个对应的 **hashcode（散列值）**
比如字符串“gareen”对应的是 1001 (实际上不是，这里是方便理解，假设的值)
比如字符串“temoo”对应的是 1004
比如字符串“db”对应的是 1008
比如字符串“annie”对应的**也是 1008**

-----保存数据-----

准备一个数组，其长度是 2000，并且设定特殊的 hashcode 算法，使得所有字符串对应的 hashcode，都会落在 0-1999 之间
要存放名字是"gareen"的英雄，就把该英雄和名称组成一个**键值对**，存放在数组的 1001 这个位置上
要存放名字是"temoo"的英雄，就把该英雄存放在数组的 1004 这个位置上
要存放名字是"db"的英雄，就把该英雄存放在数组的 1008 这个位置上
要存放名字是"annie"的英雄，然而 "annie"的 hashcode 1008 对应的位置**已经有 db 英雄了**，那么就在这里创建一个链表，**接在 db 英雄后面存放 annie**

-----查找数据-----
比如要查找 gareen，首先计算"gareen"的 hashcode 是 1001，根据 1001 这个下标，到数组中进行定位，（**根据数组下标进行定位，是非常快速的**） 发现 1001 这个位置就只有一个英雄，那么该英雄就是 gareen.
比如要查找 annie，首先计算"annie"的 hashcode 是 1008，根据 1008 这个下标，到数组中进行定位，发现 1008 这个位置**有两个英雄**，那么就对两个英雄的名字进行逐一比较(**equals**)，因为此时需要比较的量就已经少很多了，很快也就可以找出目标英雄
这就是使用 hashmap 进行查询，非常快原理。

这是一种用**空间换时间**的思维方式



HashSet 判断是否重复

HashSet 的数据是不能重复的，相同数据不能保存在一起，到底如何判断是否是重复的呢？

根据 HashSet 和 HashMap 的关系，我们了解到因为 HashSet 没有自身的实现，而是里面封装了一个 HashMap，所以本质上就是判断 HashMap 的 key 是否重复。

再通过上一步的学习，key 是否重复，是由两个步骤判断的：
hashcode 是否一样
如果 hashcode 不一样，就是在不同的坑里，一定是不重复的
如果 hashcode 一样，就是在同一个坑里，还需要进行 equals 比较
如果 equals 一样，则是重复数据
如果 equals 不一样，则是不同数据。

Comparator

假设 Hero 有三个属性 name,hp,damage
一个集合中放存放 10 个 Hero,通过 Collections.sort 对这 10 个进行排序
那么到底是 hp 小的放前面？还是 damage 小的放前面？Collections.sort 也无法确定
所以要指定到底按照哪种属性进行排序
这里就需要提供一个 Comparator 给定如何进行两个对象之间的大小比较

```
public class TestCollection {
        public static void main(String[] args) {
                Random r =new Random();
                List<Hero> heros = new ArrayList<Hero>();

                for (int i = 0; i < 10; i++) {
                        //通过随机值实例化 hero 的 hp 和 damage
                        heros.add(new Hero("hero "+ i,
r.nextInt(100), r.nextInt(100)));
                }
                System.out.println("初始化后的集合：");
                System.out.println(heros);

                //直接调用 sort 会出现编译错误，因为 Hero 有各种属性
                //到底按照哪种属性进行比较，Collections 也不知道，不
确定，所以没法排
                //Collections.sort(heros);

                //引入 Comparator，指定比较的算法
                Comparator<Hero> c = new Comparator<Hero>() {
                        @Override
                        public int compare(Hero h1, Hero h2) {
                                //按照 hp 进行排序
                                if(h1.hp>=h2.hp)
                                        return 1;    //正数表示 h1 比
h2 要大
```

```
                        else
                                return -1;
                    }
                };
                Collections.sort(heros,c);
                System.out.println("按照血量排序后的集合：");
                System.out.println(heros);
        }
}
```

Comparable

使 Hero 类实现 Comparable 接口
在类里面提供比较算法
Collections.sort 就有足够的信息进行排序了，也无需额外提供比较器 Comparator

```
public class Hero implements Comparable<Hero>{
        public String name;
        public float hp;

        public int damage;

        public Hero(){

        }

        public Hero(String name) {
                this.name =name;

        }

        //初始化 name,hp,damage 的构造方法
        public Hero(String name,float hp, int damage) {
                this.name =name;
                this.hp = hp;
                this.damage = damage;
        }

        @Override
        public int compareTo(Hero anotherHero) {
                if(damage<anotherHero.damage)
                        return 1;
                else
                        return -1;
        }
```

```
        @Override
        public String toString() {
                return "Hero [name=" + name + ", hp=" + hp + ",
damage=" + damage + "]\r\n";
        }

}
```

JDBC

创建 Statement

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
        public static void main(String[] args) {

                try {
                        Class.forName("com.mysql.jdbc.Driver");

                        Connection c = DriverManager
                                        .getConnection(
                                                        "jdbc:mysql:/
/127.0.0.1:3306/how2java?characterEncoding=UTF-8",
                                                        "root",
"admin");

                        // 注意：使用的是 java.sql.Statement
                        // 不要不小心使用到：
com.mysql.jdbc.Statement;
                        Statement s = c.createStatement();

                        System.out.println("获取 Statement 对象： " +
s);

                } catch (ClassNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
```

```
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

执行 SQL 语句

```
public class TestJDBC {
        public static void main(String[] args) {

                try {
                        Class.forName("com.mysql.jdbc.Driver");

                        Connection c = DriverManager
                                        .getConnection(
                                                        "jdbc:mysql:/
/127.0.0.1:3306/how2java?characterEncoding=UTF-8",
                                                        "root",
"admin");

                        Statement s = c.createStatement();

                        // 准备 sql 语句
                        // 注意： 字符串要用单引号'
                        String sql = "insert into hero
values(null,"+"'提莫'"+","+313.0f+","+50+")";
                        s.execute(sql);

                        System.out.println("执行插入语句成功");

                } catch (ClassNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
}
```

关闭连接

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
	public static void main(String[] args) {

		Connection c = null;
		Statement s = null;
		try {
			Class.forName("com.mysql.jdbc.Driver");

			c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how2java?cha
racterEncoding=UTF-8", "root",
					"admin");

			s = c.createStatement();

			String sql = "insert into hero values(null,"
+ "'提莫'" + "," + 313.0f + "," + 50 + ")";

			s.execute(sql);

		} catch (ClassNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (SQLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} finally {
			// 数据库的连接时有限资源，相关操作结束后，养
成关闭数据库的好习惯
			// 先关闭 Statement
			if (s != null)
				try {
					s.close();
				} catch (SQLException e) {
					// TODO Auto-generated catch
block
```

```
                                e.printStackTrace();
                        }
                    // 后关闭 Connection
                    if (c != null)
                        try {
                            c.close();
                        } catch (SQLException e) {
                            // TODO Auto-generated catch
block
                            e.printStackTrace();
                        }

                }

        }
}
```

增删改

```
String sql = "delete from hero where id = 5";
                    s.execute(sql);
```

查

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8",
                "root", "admin"); Statement s =
c.createStatement();) {
```

```
            String sql = "select * from hero";

            // 执行查询语句，并把结果集返回给 ResultSet
            ResultSet rs = s.executeQuery(sql);
            while (rs.next()) {
                int id = rs.getInt("id");// 可以使用字段名

                String name = rs.getString(2);// 也可以使用字段
的顺序

                float hp = rs.getFloat("hp");
                int damage = rs.getInt(4);
                System.out.printf("%d\t%s\t%f\t%d%n", id,
name, hp, damage);
            }
            // 不一定要在这里关闭 ReultSet，因为 Statement 关闭的
时候，会自动关闭 ResultSet
            // rs.close();

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

分页查询

设计一个方法，进行分页查询

```
public static void list(int start, int count)
```

start 表示开始页数，count 表示一页显示的总数
list(0,5) 表示第一页，一共显示 5 条数据
list(10,5) 表示第三页，一共显示 5 条数据
进行分页查询用到的 SQL 语句参考 ： 查询数据

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {

    public static void list(int start, int count){
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8",
                "root", "admin"); Statement s =
c.createStatement();) {

            String sql = "select * from hero limit " +start +
"," + count;


            // 执行查询语句，并把结果集返回给 ResultSet
            ResultSet rs = s.executeQuery(sql);
            while (rs.next()) {
                int id = rs.getInt("id");// 可以使用字段名

                String name = rs.getString(2);// 也可以使用字段
的顺序

                float hp = rs.getFloat("hp");
                int damage = rs.getInt(4);
                System.out.printf("%d\t%s\t%f\t%d%n", id,
name, hp, damage);
            }

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

```
    }

    public static void main(String[] args) {
        list(10,5);
    }
}
```

使用 PreparedStatement

和 Statement 一样，PreparedStatement 也是用来执行 sql 语句的
与创建 Statement 不同的是，需要根据 sql 语句创建 PreparedStatement
除此之外，还能能够通过设置参数，指定相应的值，而不是 Statement 那样使用字符串拼接

注： 这是 JAVA 里唯二的基 1 的地方，另一个是查询语句中的 ResultSet 也是基 1 的。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class TestJDBC {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        String sql = "insert into hero values(null,?,?,?)";
        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             // 根据 sql 语句创建 PreparedStatement
             PreparedStatement ps = c.prepareStatement(sql);
        ) {

            // 设置参数

            ps.setString(1, "提莫");
            ps.setFloat(2, 313.0f);
            ps.setInt(3, 50);
```

```
            // 执行
            ps.execute();

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

PreparedStatement 的优点 1-参数设置

Statement 需要进行字符串拼接，可读性和维护性比较差

```
String sql = "insert into hero values(null,"+"'提莫'"+","+313.0f+","+50+")";
```

PreparedStatement 使用参数设置，可读性好，不易犯错

```
String sql = "insert into hero values(null,?,?,?)";
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {

        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        String sql = "insert into hero values(null,?,?,?)";
        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             Statement s = c.createStatement();
             PreparedStatement ps = c.prepareStatement(sql);
```

```
        ) {
            // Statement 需要进行字符串拼接，可读性和维修性比较差
            String sql0 = "insert into hero values(null," +
"'提莫'" + "," + 313.0f + "," + 50 + ")";
            s.execute(sql0);

            // PreparedStatement 使用参数设置，可读性好，不易犯
错
            // "insert into hero values(null,?,?,?)";
            ps.setString(1, "提莫");
            ps.setFloat(2, 313.0f);
            ps.setInt(3, 50);
            ps.execute();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

PreparedStatement 的优点 2-性能表现

PreparedStatement 有预编译机制，性能比 Statement 更快

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {

        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
```

```
        String sql = "insert into hero values(null,?,?,?)";
        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             Statement s = c.createStatement();
             PreparedStatement ps = c.prepareStatement(sql);
        ) {
            // Statement 执行 10 次，需要 10 次把 SQL 语句传输到数据
库端

            // 数据库要对每一次来的 SQL 语句进行编译处理
            for (int i = 0; i < 10; i++) {
                String sql0 = "insert into hero values(null,"
+ "'提莫'" + ","
                        + 313.0f + "," + 50 + ")";
                s.execute(sql0);
            }
            s.close();

            // PreparedStatement 执行 10 次，只需要 1 次把 SQL 语
句传输到数据库端

            // 数据库对带?的 SQL 进行预编译


            // 每次执行，只需要传输参数到数据库端

            // 1. 网络传输量比 Statement 更小

            // 2. 数据库不需要再进行编译，相应更快
            for (int i = 0; i < 10; i++) {
                ps.setString(1, "提莫");
                ps.setFloat(2, 313.0f);
                ps.setInt(3, 50);
                ps.execute();
            }
```

```
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

PreparedStatement 的优点 3-防止 SQL 注入式攻击

假设 name 是用户提交来的数据

```
String name = "'盖伦' OR 1=1";
```

使用 Statement 就需要进行字符串拼接
拼接出来的语句是：

```
select * from hero where name = '盖伦' OR 1=1
```

因为有 OR 1=1，这是恒成立的
那么就会把所有的英雄都查出来，而不只是盖伦
如果 Hero 表里的数据时海量的，比如几百万条，**把这个表里的数据全部查出来**
会让数据库负载变高，CPU100%，内存消耗光，**响应变得极其缓慢**

而 PreparedStatement 使用的是**参数设置**，就不会有这个问题

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {

        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        String sql = "select * from hero where name = ?";
```

```
        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             Statement s = c.createStatement();
             PreparedStatement ps = c.prepareStatement(sql);
        ) {
            // 假设name是用户提交来的数据

            String name = "'盖伦' OR 1=1";

            String sql0 = "select * from hero where name = "
+ name;
            // 拼接出来的SQL语句就是

            // select * from hero where name = '盖伦' OR 1=1

            // 因为有OR 1=1，所以恒成立

            // 那么就会把所有的英雄都查出来，而不只是盖伦

            // 如果Hero表里的数据时海量的，比如几百万条，把这个
表里的数据全部查出来

            // 会让数据库负载变高，CPU100%，内存消耗光，响应变得
极其缓慢

            System.out.println(sql0);

            ResultSet rs0 = s.executeQuery(sql0);
            while (rs0.next()) {
                String heroName = rs0.getString("name");
                System.out.println(heroName);
            }

            s.execute(sql0);

            // 使用预编译Statement就可以杜绝SQL注入


            ps.setString(1, name);
```

```
            ResultSet rs = ps.executeQuery();

            // 查不出数据出来

            while (rs.next()) {
                String heroName = rs.getString("name");
                System.out.println(heroName);
            }

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

execute 与 executeUpdate 的区别

相同点

execute 与 executeUpdate 的相同点：都可以执行增加，删除，修改

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             Statement s = c.createStatement();) {

            String sqlInsert = "insert into Hero values
(null,'盖伦',616,100)";
            String sqlDelete = "delete from Hero where id =
100";
            String sqlUpdate = "update Hero set hp = 300
```

```
where id = 100";

            // 相同点：都可以执行增加，删除，修改


            s.execute(sqlInsert);
            s.execute(sqlDelete);
            s.execute(sqlUpdate);
            s.executeUpdate(sqlInsert);
            s.executeUpdate(sqlDelete);
            s.executeUpdate(sqlUpdate);

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

不同点

不同 1：
execute 可以执行查询语句
然后通过 getResultSet，把结果集取出来
executeUpdate 不能执行查询语句
不同 2:
execute 返回 boolean 类型，true 表示执行的是查询语句，false 表示执行的是 insert,delete,update 等等
executeUpdate 返回的是 int，表示有多少条数据受到了影响

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {

        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
```

```
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             Statement s = c.createStatement();) {

            // 不同1：execute 可以执行查询语句

            // 然后通过getResultSet，把结果集取出来
            String sqlSelect = "select * from hero";

            s.execute(sqlSelect);
            ResultSet rs = s.getResultSet();
            while (rs.next()) {
                System.out.println(rs.getInt("id"));
            }

            // executeUpdate 不能执行查询语句
            // s.executeUpdate(sqlSelect);

            // 不同2:

            // execute 返回boolean 类型，true 表示执行的是查询语
句，false 表示执行的是insert,delete,update 等等
            boolean isSelect = s.execute(sqlSelect);
            System.out.println(isSelect);

            // executeUpdate 返回的是int，表示有多少条数据受到了
影响
            String sqlUpdate = "update Hero set hp = 300
where id < 100";
            int number = s.executeUpdate(sqlUpdate);
            System.out.println(number);

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
```

```
        }

    }
}
```

获取自增长 id

在 Statement 通过 execute 或者 executeUpdate 执行完插入语句后，MySQL 会为新插入的数据分配一个自增长 id，(前提是这个表的 id 设置为了自增长,在 Mysql 创建表的时候，AUTO_INCREMENT 就表示自增长)

```
CREATE TABLE hero (
  id int(11) AUTO_INCREMENT,
  ...
}
```

但是无论是 execute 还是 executeUpdate 都不会返回这个自增长 id 是多少。需要通过 Statement 的 getGeneratedKeys 获取该 id
注： 第 20 行的代码，后面加了个 Statement.RETURN_GENERATED_KEYS 参数，以确保会返回自增长 ID。 通常情况下不需要加这个，有的时候需要加，所以先加上，保险一些

```
PreparedStatement ps = c.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        String sql = "insert into hero values(null,?,?,?)";
        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
```

```
2java?characterEncoding=UTF-8","root", "admin");
            PreparedStatement ps = c.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
        ) {

            ps.setString(1, "盖伦");

            ps.setFloat(2, 616);
            ps.setInt(3, 100);

            // 执行插入语句

            ps.execute();

            // 在执行完插入语句后，MySQL 会为新插入的数据分配一个
自增长 id

            // JDBC 通过 getGeneratedKeys 获取该 id

            ResultSet rs = ps.getGeneratedKeys();
            if (rs.next()) {
                int id = rs.getInt(1);
                System.out.println(id);
            }

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

获取表的元数据

元数据概念：
和数据库服务器相关的数据，比如数据库版本，有哪些表，表有哪些字段，字段类型是什么等等。

```
数据库产品名称：  MySQL
数据库产品版本：  5.5.15
数据库和表分隔符：        .
驱动版本：        mysql-connector-java-5.0.8 ( Revision: ${svn.Revision} )
可用的数据库列表：
数据库名称：      information_schema
数据库名称：      how2java
数据库名称：      mysql
数据库名称：      performance_schema
数据库名称：      test
```

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {

    public static void main(String[] args) throws Exception
{
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");) {

            // 查看数据库层面的元数据

            // 即数据库服务器版本，驱动版本，都有哪些数据库等等


            DatabaseMetaData dbmd = c.getMetaData();

            // 获取数据库服务器产品名称

            System.out.println("数据库产品名

称:\t"+dbmd.getDatabaseProductName());
```

```
            // 获取数据库服务器产品版本号
            System.out.println("数据库产品版
本:\t"+dbmd.getDatabaseProductVersion());
            // 获取数据库服务器用作类别和表名之间的分隔符 如
test.user
            System.out.println("数据库和表分隔
符:\t"+dbmd.getCatalogSeparator());
            // 获取驱动版本
            System.out.println("驱动版
本:\t"+dbmd.getDriverVersion());

            System.out.println("可用的数据库列表：");
            // 获取数据库名称
            ResultSet rs = dbmd.getCatalogs();
            while (rs.next()) {
                System.out.println("数据库名
称:\t"+rs.getString(1));
            }
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

使用事务

在事务中的多个操作，**要么都成功，要么都失败**

通过 c.setAutoCommit(false);关闭自动提交
使用 c.commit();进行手动提交
在 22 行-35 行之间的数据库操作，就处于同一个事务当中，要么都成功，要么都失败
所以，虽然第一条 SQL 语句是可以执行的，但是第二条 SQL 语句有错误，其结果就是两条 SQL 语句都没有被提交。 除非两条 SQL 语句都是正确的。

```
package jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8","root", "admin");
             Statement s = c.createStatement();) {

            // 有事务的前提下

            // 在事务中的多个操作，要么都成功，要么都失败


            c.setAutoCommit(false);

            // 加血的SQL
            String sql1 = "update hero set hp = hp +1 where
id = 22";
            s.execute(sql1);

            // 减血的SQL

            // 不小心写错写成了 updata(而非update)
```

```
            String sql2 = "updata hero set hp = hp -1 where
id = 22";
            s.execute(sql2);

            // 手动提交
            c.commit();

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

MYSQL 表的类型必须是 INNODB 才支持事务

在 Mysql 中，只有当表的类型是 INNODB 的时候，才支持事务，所以需要把表的类型设置为 INNODB,否则无法观察到事务.

修改表的类型为 INNODB 的 SQL：

```
<br data-filtered="filtered">alter table hero ENGINE    = innodb;<br data-
filtered="filtered">
```

查看表的类型的 SQL

```
<br data-filtered="filtered">show table status from how2java; <br data-
filtered="filtered">
```

不过有个前提，就是当前的 MYSQL 服务器本身要支持 INNODB,如果不支持，请看 开启 MYSQL INNODB 的办法

练习-事务

当 **c.setAutoCommit(false);**时，事务是不会提交的
只有执行使用 **c.commit();** 才会提交进行

设计一个代码，删除表中前 10 条数据，但是删除前会在控制台弹出一个提示：
是否要删除数据(Y/N)
如果用户输入 Y，则删除
如果输入 N 则不删除。
如果输入的既不是 Y 也不是 N，则重复提示

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class TestJDBC {

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        try (Connection c =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/how
2java?characterEncoding=UTF-8",
                "root", "admin");
             Statement st4Query = c.createStatement();
             Statement st4Delete = c.createStatement();
             Scanner s = new Scanner(System.in);) {


            //把自动提交关闭

            c.setAutoCommit(false);

            //查出前10条

            ResultSet rs =st4Query.executeQuery("select id
from Hero order by id asc limit 0,10 ");
            while(rs.next()){
                int id = rs.getInt(1);

                System.out.println("试图删除 id="+id+" 的数据

");
```

```
                st4Delete.execute("delete from Hero where id
= " +id);
            }

            //是否删除这10条
            while(true){
                System.out.println("是否要删除数据(Y/N)");

                String str = s.next();
                if ("Y".equals(str)) {
                    //如果输入的是Y，则提交删除操作
                    c.commit();
                    System.out.println("提交删除");
                    break;
                } else if ("N".equals(str)) {
                    System.out.println("放弃删除");
                    break;
                }
            }
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

多线程

创建多线程有 3 种方式，分别是继承线程类,实现 Runnable 接口,匿名类

当前线程暂停

Thread.sleep(1000); 表示当前线程暂停 1000 毫秒 ， 其他线程不受影响
Thread.sleep(1000); 会抛出 InterruptedException 中断异常，因为当前线程 sleep 的时候，有可能被停止，这时就会抛出 InterruptedException

```
public class TestThread {

    public static void main(String[] args) {

        Thread t1= new Thread(){
            public void run(){
                int seconds =0;
                while(true){
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    System.out.printf("已经玩了 LOL %d 秒%n",
seconds++);
                }
            }
        };
        t1.start();

    }

}
```

加入到当前线程中

首先解释一下主线程的概念
所有进程，至少会有一个线程即主线程，即 main 方法开始执行，就会有一个看不见的主线程存在。
在 44 行执行 t.join，即表明在主线程中加入该线程。
主线程会等待该线程结束完毕， 才会往下运行。

```
public class TestThread {

    public static void main(String[] args) {

        final Hero gareen = new Hero();

        gareen.name = "盖伦";

        gareen.hp = 616;
        gareen.damage = 50;

        final Hero teemo = new Hero();
```

```
        teemo.name = "提莫";
        teemo.hp = 300;
        teemo.damage = 30;

        final Hero bh = new Hero();
        bh.name = "赏金猎人";
        bh.hp = 500;
        bh.damage = 65;

        final Hero leesin = new Hero();
        leesin.name = "盲僧";
        leesin.hp = 455;
        leesin.damage = 80;

        Thread t1= new Thread(){
            public void run(){
                while(!teemo.isDead()){
                    gareen.attackHero(teemo);
                }
            }
        };

        t1.start();

        //代码执行到这里，一直是main 线程在运行
        try {
            //t1 线程加入到main 线程中来，只有t1 线程运行结束，才
会继续往下走
            t1.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        Thread t2= new Thread(){
            public void run(){
                while(!leesin.isDead()){
                    bh.attackHero(leesin);
```

```
                }
            }
        };
        //会观察到盖伦把提莫杀掉后，才运行 t2 线程
        t2.start();

    }

}
```

线程优先级

当线程处于竞争关系的时候，优先级高的线程会有更大的几率获得 CPU 资源
为了演示该效果，要把暂停时间去掉，多条线程各自会尽力去占有 CPU 资源
同时把英雄的血量增加 100 倍，攻击减低到 1，才有足够的时间观察到优先级的演示
如图可见，线程 1 的优先级是 MAX_PRIORITY，所以它争取到了更多的 CPU 资源
执行代码

```
t1.setPriority(Thread.MAX_PRIORITY);
```

临时暂停

当前线程，临时暂停，使得其他线程可以有更多的机会占用 CPU 资源

```
//临时暂停，使得 t1 可以占用 CPU 资源
                                        Thread.yield();
```

守护线程

守护线程的概念是： 当一个进程里，所有的线程都是守护线程的时候，结束当前进程。

就好像一个公司有销售部，生产部这些和业务挂钩的部门。
除此之外，还有后勤，行政等这些支持部门。

如果一家公司销售部，生产部都解散了，那么只剩下后勤和行政，那么这家公司也可以解散了。

守护线程就相当于那些支持部门，如果一个进程只剩下守护线程，那么进程就会自动结束。

守护线程通常会被用来做日志，性能统计等工作。

```
public class TestThread {

    public static void main(String[] args) {

        Thread t1= new Thread(){
            public void run(){
                int seconds =0;

                while(true){
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    System.out.printf("已经玩了 LOL %d 秒%n",
seconds++);

                }
            }
        };
        t1.setDaemon(true);
        t1.start();

    }

}
```

练习-破解密码

1. 生成一个长度是 3 的随机字符串，把这个字符串作为当做密码

2. 创建一个破解线程，使用穷举法，匹配这个密码

3. 创建一个日志线程，打印都用过哪些字符串去匹配，这个日志线程设计为守护线程

提示： 破解线程把穷举法生成的可能密码放在一个容器中，日志线程不断的从这个容器中拿出可能密码，并打印出来。 如果发现容器是空的，就休息 1 秒，如果发现不是空的，就不停的取出，并打印。

穷举密码的线程

```
package multiplethread;

import java.util.List;

public class PasswordThread extends Thread{

    private boolean found = false;

    private String password;

    private List<String> passwords;

    public PasswordThread(String password, List<String>
passwords) {
        this.password = password;
        this.passwords = passwords;
    }

    public void run(){
        char[] guessPassword = new char[password.length()];
        generatePassword(guessPassword, password);
    }

    public  void generatePassword(char[] guessPassword,
String password) {
        generatePassword(guessPassword, 0, password);
    }

    public  void generatePassword(char[] guessPassword, int
index, String password) {
        if (found)
            return;
        for (short i = '0'; i <= 'z'; i++) {
            char c = (char) i;
            if (!Character.isLetterOrDigit(c))
                continue;
            guessPassword[index] = c;
            if (index != guessPassword.length - 1) {
                generatePassword(guessPassword, index + 1,
password);
            } else {
                String guess = new String(guessPassword);
                //穷举每次生成的密码，都放进集合中
```

```
                passwords.add(guess);
                if (guess.equals(password)) {
                    System.out.println("找到了，密码是" +
guess);
                    found = true;
                    return;
                }

            }
        }
    }

}
```

记录日志的守护线程

```
package multiplethread;

import java.util.List;

public class LogThread extends Thread{
    private boolean found = false;

    private List<String> passwords;

    public LogThread(List<String> passwords) {
        this.passwords = passwords;


        this.setDaemon(true);//把记日志的这个线程，设置为守护线
程

    }

    public void run(){
        while(true){
            while(passwords.isEmpty()){
                try {
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
```

```
            String password = passwords.remove(0);
            System.out.println("穷举法本次生成的密码是：" +password);
        }
    }

}
```

测试

```
package multiplethread;

import java.util.ArrayList;
import java.util.List;

public class TestThread {

    public static boolean found = false;

    public static void main(String[] args) {
        String password = randomString(3);
        System.out.println("密码是:" + password);
        List<String> passwords = new ArrayList<>();

        new PasswordThread(password,passwords).start();
        new LogThread(passwords).start();

    }

    private static String randomString(int length) {
        String pool = "";
        for (short i = '0'; i <= '9'; i++) {
            pool += (char) i;
        }
        for (short i = 'a'; i <= 'z'; i++) {
            pool += (char) i;
        }
        for (short i = 'A'; i <= 'Z'; i++) {
            pool += (char) i;
        }
        char cs[] = new char[length];
        for (int i = 0; i < cs.length; i++) {
            int index = (int) (Math.random() *
```

```
pool.length());
            cs[i] = pool.charAt(index);
        }
        String result = new String(cs);
        return result;
    }
}
```

线程同步

解决思路

总体解决思路是： 在增加线程访问 hp 期间，其他线程不可以访问 hp
1. 增加线程获取到 hp 的值，并进行运算
2. 在运算期间，减少线程试图来获取 hp 的值，但是不被允许
3. 增加线程运算结束，并成功修改 hp 的值为 10001
4. 减少线程，在增加线程做完后，才能访问 hp 的值，即 10001
5. 减少线程运算，并得到新的值 10000

synchronized 同步对象概念

解决上述问题之前，先理解
synchronized 关键字的意义
如下代码：

```
Object someObject =new Object();
synchronized (someObject){
   //此处的代码只有占有了 someObject 后才可以执行
}
```

synchronized 表示当前线程，独占 对象 someObject
当前线程独占 了对象 someObject，如果有其他线程试图占有对象 someObject，就会等待，直到当前线程释放对 someObject 的占用。
someObject 又叫同步对象，所有的对象，都可以作为同步对象
为了达到同步的效果，必须使用同一个同步对象

释放同步对象的方式： synchronized 块自然结束，或者有异常抛出

```
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
public class TestThread {

    public static String now(){
        return new SimpleDateFormat("HH:mm:ss").format(new
Date());
    }

    public static void main(String[] args) {
        final Object someObject = new Object();

        Thread t1 = new Thread(){
            public void run(){
                try {
                    System.out.println( now()+" t1 线程已经运行
");
                    System.out.println( now()+this.getName()+
" 试图占有对象：someObject");
                    synchronized (someObject) {


System.out.println( now()+this.getName()+ " 占有对象：
someObject");
                        Thread.sleep(5000);

System.out.println( now()+this.getName()+ " 释放对象：
someObject");
                    }
                    System.out.println(now()+" t1 线程结束");
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };
        t1.setName(" t1");
        t1.start();
        Thread t2 = new Thread(){

            public void run(){
```

```
                try {
                    System.out.println( now()+" t2 线程已经运行
");
                    System.out.println( now()+this.getName()+
" 试图占有对象：someObject");
                    synchronized (someObject) {

System.out.println( now()+this.getName()+ " 占有对象：
someObject");
                        Thread.sleep(5000);

System.out.println( now()+this.getName()+ " 释放对象：
someObject");
                    }
                    System.out.println(now()+" t2 线程结束");
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };
        t2.setName(" t2");
        t2.start();
    }

}
```

使用 synchronized 解决同步问题

所有需要修改 hp 的地方，有要**建立在占有 someObject 的基础上**。
而对象 someObject 在同一时间，只能被一个线程占有。 间接地，**导致同一时间，hp 只能被一个线程修改。**

```
import java.awt.GradientPaint;

import charactor.Hero;

public class TestThread {

    public static void main(String[] args) {
```

```
        final Object someObject = new Object();

        final Hero gareen = new Hero();
        gareen.name = "盖伦";
        gareen.hp = 10000;

        int n = 10000;

        Thread[] addThreads = new Thread[n];
        Thread[] reduceThreads = new Thread[n];

        for (int i = 0; i < n; i++) {
            Thread t = new Thread(){
                public void run(){

                    //任何线程要修改hp的值，必须先占用
someObject
                    synchronized (someObject) {
                        gareen.recover();
                    }

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            t.start();
            addThreads[i] = t;

        }

        for (int i = 0; i < n; i++) {
            Thread t = new Thread(){
                public void run(){
                    //任何线程要修改hp的值，必须先占用
someObject
                    synchronized (someObject) {
```

```
                        gareen.hurt();
                    }

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            t.start();
            reduceThreads[i] = t;
        }

        for (Thread t : addThreads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        for (Thread t : reduceThreads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }


        System.out.printf("%d 个增加线程和%d 个减少线程结束后%n
盖伦的血量是 %.0f%n", n,n,gareen.hp);

    }

}
```

使用 hero 对象作为同步对象

既然任意对象都可以用来作为同步对象，而所有的线程访问的都是同一个 hero 对

象，**索性就使用 gareen 来作为同步对象**
进一步的，对于 Hero 的 hurt 方法，加上：
synchronized (this) {
}
表示当期对象为同步对象，即也是 gareen 为同步对象

```
public class TestThread {

    public static void main(String[] args) {

        final Hero gareen = new Hero();

        gareen.name = "盖伦";

        gareen.hp = 10000;

        int n = 10000;

        Thread[] addThreads = new Thread[n];
        Thread[] reduceThreads = new Thread[n];

        for (int i = 0; i < n; i++) {
            Thread t = new Thread(){
                public void run(){

                    //使用gareen 作为synchronized

                    synchronized (gareen) {
                        gareen.recover();
                    }

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            t.start();
            addThreads[i] = t;

        }

        for (int i = 0; i < n; i++) {
            Thread t = new Thread(){
```

```
                public void run(){
                    //使用gareen 作为synchronized

                    //在方法hurt 中有synchronized(this)
                    gareen.hurt();

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            t.start();
            reduceThreads[i] = t;
        }

        for (Thread t : addThreads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        for (Thread t : reduceThreads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        System.out.printf("%d 个增加线程和%d 个减少线程结束后%n
盖伦的血量是 %.0f%n", n,n,gareen.hp);

    }

}
```

```
package charactor;

public class Hero{
    public String name;
    public float hp;

    public int damage;


    //回血
    public void recover(){
        hp=hp+1;
    }


    //掉血
    public void hurt(){
        //使用this 作为同步对象
        synchronized (this) {
            hp=hp-1;
        }
    }

    public void attackHero(Hero h) {
        h.hp-=damage;
        System.out.format("%s 正在攻击 %s, %s 的血变成了 %.0f%n",name,h.name,h.name,h.hp);
        if(h.isDead())
            System.out.println(h.name +"死了！ ");
    }

    public boolean isDead() {
        return 0>=hp?true:false;
    }

}
```

在方法前，加上修饰符 synchronized

在 recover 前，直接加上 synchronized ，其所对应的同步对象，就是 this

和 hurt 方法达到的效果是一样
外部线程访问 gareen 的方法，就不需要额外使用 synchronized 了

```
public class Hero{
    public String name;
    public float hp;

    public int damage;


    //回血

    //直接在方法前加上修饰符 synchronized

    //其所对应的同步对象，就是 this

    //和 hurt 方法达到的效果一样
    public synchronized void recover(){
        hp=hp+1;
    }


    //掉血
    public void hurt(){
        //使用 this 作为同步对象
        synchronized (this) {
            hp=hp-1;
        }
    }

    public void attackHero(Hero h) {
        h.hp-=damage;
        System.out.format("%s 正在攻击 %s, %s 的血变成了 %.0f%n",name,h.name,h.name,h.hp);
        if(h.isDead())
            System.out.println(h.name +"死了！ ");
    }

    public boolean isDead() {
        return 0>=hp?true:false;
    }
```

```
}

import java.awt.GradientPaint;

import charactor.Hero;

public class TestThread {

    public static void main(String[] args) {

        final Hero gareen = new Hero();

        gareen.name = "盖伦";

        gareen.hp = 10000;

        int n = 10000;

        Thread[] addThreads = new Thread[n];
        Thread[] reduceThreads = new Thread[n];

        for (int i = 0; i < n; i++) {
            Thread t = new Thread(){
                public void run(){

                    //recover 自带 synchronized

                    gareen.recover();

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            t.start();
            addThreads[i] = t;

        }

        for (int i = 0; i < n; i++) {
            Thread t = new Thread(){
```

```
                public void run(){
                    //hurt 自带 synchronized
                    gareen.hurt();

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            t.start();
            reduceThreads[i] = t;
        }

        for (Thread t : addThreads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        for (Thread t : reduceThreads) {
            try {
                t.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        System.out.printf("%d 个增加线程和%d 个减少线程结束后%n
盖伦的血量是 %.0f%n", n,n,gareen.hp);

    }

}
```

线程安全的类

如果一个类，其**方法都是有 synchronized 修饰的**，那么该类就叫做**线程安全的类**

同一时间，只有一个线程能够进入 **这种类的一个实例** 的去修改数据，进而保证了这个实例中的数据的安全(不会同时被多线程修改而变成脏数据)

比如 StringBuffer 和 StringBuilder 的区别
StringBuffer 的方法都是有 synchronized 修饰的，StringBuffer 就叫做线程安全的类
而 StringBuilder 就不是线程安全的类

演示死锁

1. 线程 1 首先占有对象 1，接着试图占有对象 2
2. 线程 2 首先占有对象 2，接着试图占有对象 1
3. 线程 1 等待线程 2 释放对象 2
4. 与此同时，线程 2 等待线程 1 释放对象 1
就会。。。一直等待下去，直到天荒地老，海枯石烂，山无棱 ， 天地合。。。

```
public class TestThread {

    public static void main(String[] args) {
        final Hero ahri = new Hero();

        ahri.name = "九尾妖狐";

        final Hero annie = new Hero();

        annie.name = "安妮";


        Thread t1 = new Thread(){
            public void run(){

                //占有九尾妖狐

                synchronized (ahri) {

                    System.out.println("t1 已占有九尾妖狐");

                    try {

                        //停顿1000秒，另一个线程有足够的时间占有

安妮


                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
```

```
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }


                System.out.println("t1 试图占有安妮");

                System.out.println("t1 等待中 。。。。");
                synchronized (annie) {
                    System.out.println("do something");
                }
            }

        }
    };
    t1.start();
    Thread t2 = new Thread(){
        public void run(){
            //占有安妮
            synchronized (annie) {
                System.out.println("t2 已占有安妮");
                try {

                    //停顿1000秒，另一个线程有足够的时间占有
暂用九尾妖狐
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                System.out.println("t2 试图占有九尾妖狐");

                System.out.println("t2 等待中 。。。。");
                synchronized (ahri) {
                    System.out.println("do something");
                }
            }

        }
```

```
        };
        t2.start();
    }

}
```

练习-死锁

3 个同步对象 a, b, c
3 个线程 t1,t2,t3

故意设计场景，使这 3 个线程彼此死锁

```
public class TestThread {

    public static void main(String[] args) {
        Object a = new Object();
        Object b = new Object();
        Object c = new Object();

        Thread t1 =new Thread(){
            public void run(){
                synchronized (a) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    synchronized (b) {
                        synchronized (c) {

                        }
                    }
                }
            }
        };
        Thread t2 =new Thread(){
            public void run(){
                synchronized (c) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
                }
                synchronized (a) {
                    synchronized (b) {

                    }
                }
            }
        }
    };
    Thread t3 =new Thread(){
        public void run(){
            synchronized (b) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                synchronized (c) {
                    synchronized (a) {

                    }
                }
            }
        }
    };

    t1.start();
    t2.start();
    t3.start();

}

}
```

交互

线程之间有**交互通知**的需求，考虑如下情况：
有两个线程，处理同一个英雄。
一个加血，一个减血。

减血的线程，发现血量=1，就停止减血，直到加血的线程为英雄加了血，才可以继续减血

不好的解决方式

故意设计减血线程频率更高，盖伦的血量迟早会到达 1
减血线程中**使用 while 循环判断是否是 1**，如果是 1 就不停的循环,直到加血线程回复了血量
这是不好的解决方式，因为会大量占用 CPU,拖慢性能

```
public class Hero{
    public String name;
    public float hp;

    public int damage;

    public synchronized void recover(){
        hp=hp+1;
    }

    public synchronized void hurt(){
        hp=hp-1;
    }

    public void attackHero(Hero h) {
        h.hp-=damage;

        System.out.format("%s 正在攻击 %s, %s 的血变成

了 %.0f%n",name,h.name,h.name,h.hp);

        if(h.isDead())

            System.out.println(h.name +"死了！ ");

    }

    public boolean isDead() {
        return 0>=hp?true:false;
    }

}
package multiplethread;

import java.awt.GradientPaint;

import charactor.Hero;

public class TestThread {
```

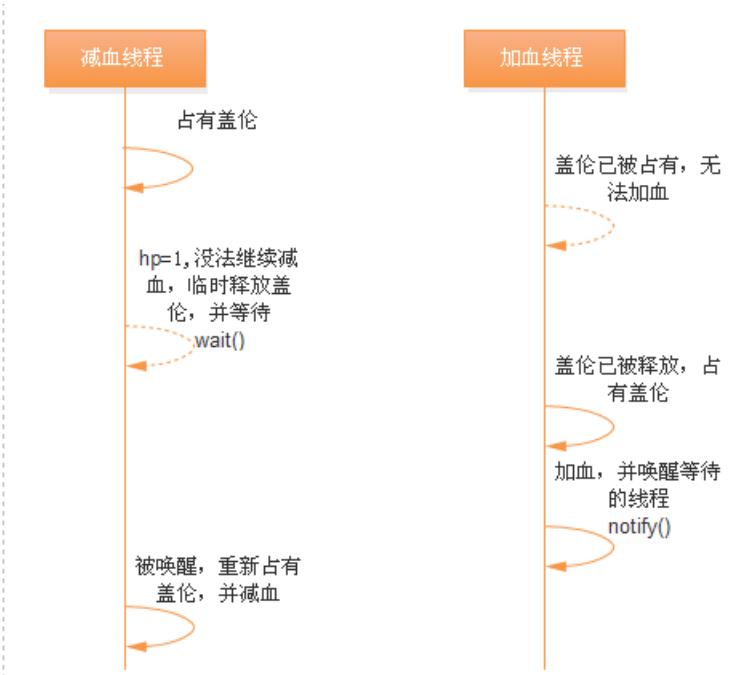
```
    public static void main(String[] args) {

        final Hero gareen = new Hero();
        gareen.name = "盖伦";
        gareen.hp = 616;

        Thread t1 = new Thread(){
            public void run(){
                while(true){

                    //因为减血更快，所以盖伦的血量迟早会到达1

                    //使用while循环判断是否是1，如果是1就不停的
循环

                    //直到加血线程回复了血量
                    while(gareen.hp==1){
                        continue;
                    }

                    gareen.hurt();
                    System.out.printf("t1 为%s 减血1点,减少血
后，%s 的血量是%.0f%n",gareen.name,gareen.name,gareen.hp);
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

            }
        };
        t1.start();

        Thread t2 = new Thread(){
            public void run(){
                while(true){
```

```
                    gareen.recover();
                    System.out.printf("t2 为%s 回血 1 点,增加血
后，%s 的血量是%.0f%n",gareen.name,gareen.name,gareen.hp);

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

            }
        };
        t2.start();

    }

}
```

使用 wait 和 notify 进行线程交互

在 Hero 类中：hurt()减血方法：当 hp=1 的时候，执行 this.wait().
this.wait()表示 让占有 this 的线程等待，并临时释放占有
进入 hurt 方法的线程必然是减血线程，this.wait()会让减血线程临时释放对 this 的占有。 这样加血线程，就有机会进入 recover()加血方法了。

recover() 加血方法：增加了血量，执行 this.notify();
this.notify() 表示通知那些等待在 this 的线程，可以苏醒过来了。 等待在 this 的线程，恰恰就是减血线程。 一旦 recover()结束， 加血线程释放了 this，减血线程，就可以重新占有 this，并执行后面的减血工作。




```
public class Hero {
    public String name;
    public float hp;

    public int damage;

    public synchronized void recover() {
        hp = hp + 1;

        System.out.printf("%s 回血 1 点,增加血后，%s 的血量

是%.0f%n", name, name, hp);

        // 通知那些等待在 this 对象上的线程，可以醒过来了，如第 20

行，等待着的减血线程，苏醒过来

        this.notify();
    }

    public synchronized void hurt() {
        if (hp == 1) {
            try {

                // 让占有 this 的减血线程，暂时释放对 this 的占有，

并等待

                this.wait();
```

```
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        hp = hp - 1;

        System.out.printf("%s 减血 1 点,减少血后，%s 的血量
是%.0f%n", name, name, hp);
    }

    public void attackHero(Hero h) {
        h.hp -= damage;

        System.out.format("%s 正在攻击 %s, %s 的血变成
了 %.0f%n", name, h.name, h.name, h.hp);

        if (h.isDead())

            System.out.println(h.name + "死了！");

    }

    public boolean isDead() {
        return 0 >= hp ? true : false;
    }

}
import java.awt.GradientPaint;

import charactor.Hero;

public class TestThread {

    public static void main(String[] args) {

        final Hero gareen = new Hero();

        gareen.name = "盖伦";

        gareen.hp = 616;

        Thread t1 = new Thread(){
            public void run(){
```

```
                while(true){

                    //无需循环判断
//                    while(gareen.hp==1){
//                        continue;
//                    }

                    gareen.hurt();

                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

            }
        };
        t1.start();

        Thread t2 = new Thread(){
            public void run(){
                while(true){
                    gareen.recover();

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

            }
        };
        t2.start();

    }

}
```

关于 wait、notify 和 notifyAll

留意 wait()和 notify() 这两个方法是什么对象上的？

```
<br data-filtered="filtered">public synchronized void hurt() {<br data-filtered="filtered">    。。。<br data-filtered="filtered">    this.wait();<br data-filtered="filtered">    。。。<br data-filtered="filtered">}<br data-filtered="filtered">
```

```
<br data-filtered="filtered">public synchronized void recover() {<br data-filtered="filtered">    。。。<br data-filtered="filtered">    this.notify();<br data-filtered="filtered">}<br data-filtered="filtered">
```

这里需要强调的是，wait 方法和 notify 方法，并不是 Thread 线程上的方法，它们是 Object 上的方法。

因为所有的 Object 都可以被用来作为同步对象，所以准确的讲，wait 和 notify 是同步对象上的方法。

wait()的意思是： 让占用了这个同步对象的线程，临时释放当前的占用，并且等待。所以调用 wait 是有前提条件的，一定是在 synchronized 块里，否则就会出错。

notify() 的意思是，通知一个等待在这个同步对象上的线程，你可以苏醒过来了，有机会重新占用当前对象了。

notifyAll() 的意思是，通知所有的等待在这个同步对象上的线程，你们可以苏醒过来了，有机会重新占用当前对象了。

练习-线程交互

假设加血线程运行得更加频繁，英雄的最大血量是 1000

设计加血线程和减血线程的交互，让回血回满之后，加血线程等待，直到有减血线程减血

```
public class Hero {
    public String name;
    public float hp;

    public int damage;
```

```
    public synchronized void recover() {
        //当血量大于或者等于1000的时候

        //this.wait() 让占用这个对象的线程等待，并临时释放锁
        if(hp>=1000){
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        hp = hp + 1;

        System.out.printf("%s 回血1点,增加血后，%s 的血量
是%.0f%n", name, name, hp);

        this.notify();
    }

    public synchronized void hurt() {
        if (hp == 1) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        hp = hp - 1;

        System.out.printf("%s 减血1点,减少血后，%s 的血量
是%.0f%n", name, name, hp);

        //掉血之后，唤醒等待的线程

        this.notify();
    }

    public void attackHero(Hero h) {
```

```
        h.hp -= damage;
        System.out.format("%s 正在攻击 %s, %s 的血变成
了 %.0f%n", name, h.name, h.name, h.hp);
        if (h.isDead())
            System.out.println(h.name + "死了！");
    }

    public boolean isDead() {
        return 0 >= hp ? true : false;
    }

}
public class TestThread {

    public static void main(String[] args) {

        final Hero gareen = new Hero();
        gareen.name = "盖伦";
        gareen.hp = 616;

        Thread t1 = new Thread(){
            public void run(){
                while(true){
                    gareen.hurt();

                    try {
                        //减慢掉血的速度
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

            }
        };
        t1.start();

        Thread t2 = new Thread(){
```

```
            public void run(){
                while(true){
                    gareen.recover();

                    try {
                        //加快回血的速度
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }

            }
        };
        t2.start();

    }

}
```

练习-多线程交互

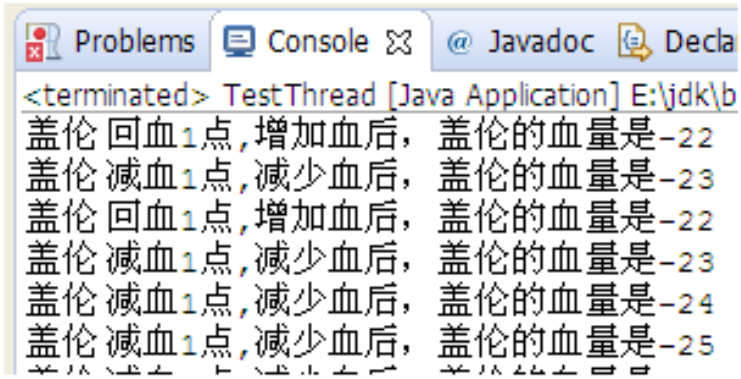
在上面的练习的基础上，增加回血线程到 2 条，减血线程到 5 条，同时运行。

运行一段时间，观察会发生的错误，分析错误原因，并考虑解决办法
在目前的状态下，会导致英雄的血量变为负数。 这是因为减血线程调用 hurt() 方法结束时，调用 notify，有可能会唤醒另一个减血线程，这就导致不停的减血。

解决办法是： 减血线程被唤醒后，要再次查看当前血量，如果当前血量<=1,那么就继续等待

//把 if 改为 while，被唤醒后，会重复查看 hp 的值，只有 hp 大于 1，才会往下执行

```
<br data-filtered="filtered"><br data-filtered="filtered">//把 if 改为 while，被唤醒后，会重复查看 hp 的值，只有 hp 大于 1，才会往下执行
```




```
public class Hero {
    public String name;
    public float hp;

    public int damage;

    public synchronized void recover() {
        //当血量大于或者等于1000 的时候

        //this.wait() 让占用这个对象的线程等待，并临时释放锁
        while(hp>=1000){
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        hp = hp + 1;

        System.out.printf("%s 回血 1 点,增加血后，%s 的血量
是%.0f%n", name, name, hp);

        this.notify();
    }

    public synchronized void hurt() {
        //把 if 改为while，被唤醒后，会重复查看 hp 的值，只有 hp 大
于1，才会往下执行
```

```
        //if (hp <= 1) {
        while (hp <= 1) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        hp = hp - 1;

        System.out.printf("%s 减血 1 点,减少血后，%s 的血量
是%.0f%n", name, name, hp);

        //掉血之后，唤醒等待的线程

        this.notify();
    }

    public void attackHero(Hero h) {
        h.hp -= damage;

        System.out.format("%s 正在攻击 %s, %s 的血变成
了 %.0f%n", name, h.name, h.name, h.hp);

        if (h.isDead())

            System.out.println(h.name + "死了！");

    }

    public boolean isDead() {
        return 0 >= hp ? true : false;
    }

}
import charactor.Hero;

public class TestThread {

    static class HurtThread extends Thread{
        private Hero hero;

        public HurtThread(Hero hero){
```

```
            this.hero = hero;
        }

        public void run(){
            while(true){
                hero.hurt();
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

    }
    static class RecoverThread extends Thread{
        private Hero hero;

        public RecoverThread(Hero hero){
            this.hero = hero;
        }

        public void run(){
            while(true){
                hero.recover();
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

    }

    public static void main(String[] args) {
        final Hero gareen = new Hero();

        gareen.name = "盖伦";

        gareen.hp = 616;

        for (int i = 0; i < 2; i++) {
```

```
            new RecoverThread(gareen).start();
        }
        for (int i = 0; i < 5; i++) {
            new HurtThread(gareen).start();
        }

    }

}
```

练习-生产者消费者问题

生产者消费者问题是一个非常典型性的线程交互的问题。

1. 使用栈来存放数据
1.1 把栈改造为支持线程安全
1.2 把栈的边界操作进行处理，当栈里的数据是 0 的时候，访问 pull 的线程就会等待。 当栈里的数据时 200 的时候，访问 push 的线程就会等待
2. 提供一个生产者（Producer）线程类，生产随机大写字符压入到堆栈
3. 提供一个消费者（Consumer）线程类，从堆栈中弹出字符并打印到控制台
4. 提供一个测试类，使两个生产者和三个消费者线程同时运行，结果类似如下 ：

```
<terminated> TestThread [Jav
Consumer3 弹出: C
Producer1 压入: F
Consumer2 弹出: F
Producer2 压入: X
Consumer2 弹出: X
Producer1 压入: A
Consumer3 弹出: A
Producer1 压入: Z
Consumer2 弹出: Z
Producer2 压入: Z
Consumer1 弹出: Z
```

```
package multiplethread;

import java.util.ArrayList;
import java.util.LinkedList;

public class MyStack<T> {

    LinkedList<T> values = new LinkedList<T>();
```

```
    public synchronized void push(T t) {
        while(values.size()>=200){
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        this.notifyAll();
        values.addLast(t);

    }

    public synchronized T pull() {
        while(values.isEmpty()){
            try {
                this.wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        this.notifyAll();
        return values.removeLast();
    }

    public T peek() {
        return values.getLast();
    }
}
package multiplethread;

public class ProducerThread extends Thread{

    private MyStack<Character> stack;

    public ProducerThread(MyStack<Character> stack,String
name){
        super(name);
        this.stack =stack;
    }
```

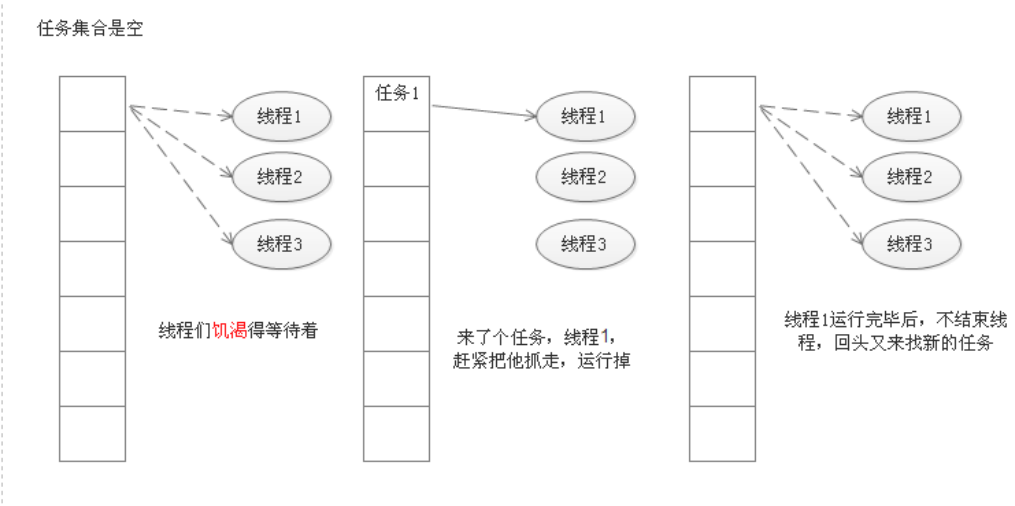
```
    public void run(){

        while(true){
            char c = randomChar();

            System.out.println(this.getName()+" 压入: " + c);

            stack.push(c);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

    }

    public char randomChar(){
        return (char) (Math.random()*('Z'+1-'A') + 'A');
    }

}
package multiplethread;

public class ConsumerThread extends Thread{

    private MyStack<Character> stack;

    public ConsumerThread(MyStack<Character> stack,String
name){
        super(name);
        this.stack =stack;
    }

    public void run(){

        while(true){
            char c = stack.pull();

            System.out.println(this.getName()+" 弹出: " + c);

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
```

```
                e.printStackTrace();
            }
        }

    }

    public char randomChar(){
        return (char) (Math.random()*('Z'+1-'A') + 'A');
    }

}
public class TestThread {

    public static void main(String[] args) {
        MyStack<Character> stack = new MyStack<>();
        new ProducerThread(stack, "Producer1").start();
        new ProducerThread(stack, "Producer2").start();
        new ConsumerThread(stack, "Consumer1").start();
        new ConsumerThread(stack, "Consumer2").start();
        new ConsumerThread(stack, "Consumer3").start();

    }

}
```

线程池设计思路

线程池的思路和生产者消费者模型是很接近的。
1. 准备一个任务容器
2. 一次性启动 10 个 消费者线程
3. 刚开始任务容器是空的，所以线程都 **wait** 在上面。
4. 直到一个外部线程往这个任务容器中扔了一个“任务”，就会有一个消费者线程被唤醒 notify
5. 这个消费者线程取出“任务”，并且**执行这个任务**，执行完毕后，继续等待下一次任务的到来。
6. 如果短时间内，有较多的任务加入，那么就会有多个线程被**唤醒**，去执行这些任务。

在整个过程中，都不需要创建新的线程，而是**循环使用这些已经存在的线程**



开发一个自定义线程池

这是一个自定义的线程池，虽然不够完善和健壮，但是已经足以说明线程池的工作原理

缓慢的给这个线程池添加任务，会看到有多条线程来执行这些任务。
线程 7 执行完毕任务后，又回到池子里，下一次任务来的时候，线程 7 又来执行新的任务。

```
<terminated> ThreadPool [Java Application] E:\jdk\bin\jav
启动:  任务消费者线程 1
启动:  任务消费者线程 2
启动:  任务消费者线程 0
启动:  任务消费者线程 4
启动:  任务消费者线程 6
启动:  任务消费者线程 8
启动:  任务消费者线程 3
启动:  任务消费者线程 5
启动:  任务消费者线程 7
启动:  任务消费者线程 9
任务消费者线程 5 获取到任务，并执行
任务消费者线程 7 获取到任务，并执行
任务消费者线程 7 获取到任务，并执行
任务消费者线程 7 获取到任务，并执行
任务消费者线程 7 获取到任务，并执行
```

```
package multiplethread;

import java.util.LinkedList;

public class ThreadPool {
```

```
    // 线程池大小
    int threadPoolSize;

    // 任务容器
    LinkedList<Runnable> tasks = new LinkedList<Runnable>();

    // 试图消费任务的线程


    public ThreadPool() {
        threadPoolSize = 10;

        // 启动10 个任务消费者线程
        synchronized (tasks) {
            for (int i = 0; i < threadPoolSize; i++) {
                new TaskConsumeThread("任务消费者线程 " +
i).start();
            }
        }
    }

    public void add(Runnable r) {
        synchronized (tasks) {
            tasks.add(r);
            // 唤醒等待的任务消费者线程
            tasks.notifyAll();
        }
    }

    class TaskConsumeThread extends Thread {
        public TaskConsumeThread(String name) {
            super(name);
        }

        Runnable task;

        public void run() {
            System.out.println("启动:  " + this.getName());
```

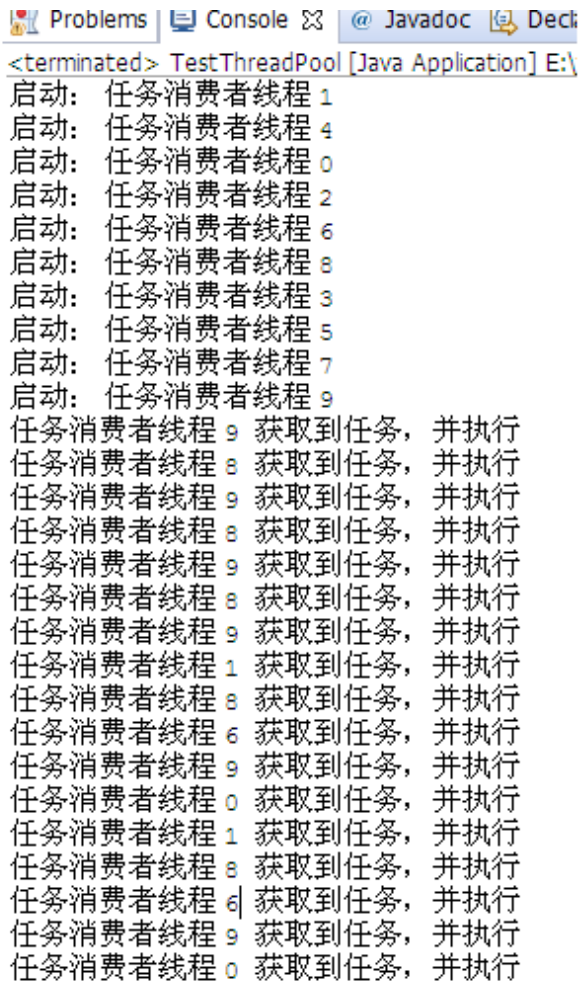
```
            while (true) {
                synchronized (tasks) {
                    while (tasks.isEmpty()) {
                        try {
                            tasks.wait();
                        } catch (InterruptedException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                    }
                    task = tasks.removeLast();

                    // 允许添加任务的线程可以继续添加任务

                    tasks.notifyAll();

                }

                System.out.println(this.getName() + " 获取到任
务，并执行");

                task.run();
            }
        }
    }

}

package multiplethread;

public class TestThread {

    public static void main(String[] args) {
        ThreadPool pool = new ThreadPool();

        for (int i = 0; i < 5; i++) {
            Runnable task = new Runnable() {
                @Override
                public void run() {

                    //System.out.println("执行任务");

                    //任务可能是打印一句话

                    //可能是访问文件
```

```
                //可能是做排序
            }
        };

        pool.add(task);

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}

}
```

测试线程池

创造一个情景，每个任务执行的时间都是 1 秒
刚开始是间隔 1 秒钟向线程池中添加任务

然后间隔时间越来越短，执行任务的线程还没有来得及结束，新的任务又来了。
就会观察到线程池里的其他线程被唤醒来执行这些任务




```java
package multiplethread;

public class TestThread {
    public static void main(String[] args) {
        ThreadPool pool= new ThreadPool();
        int sleep=1000;
        while(true){
            pool.add(new Runnable(){
                @Override
                public void run() {

                    //System.out.println("执行任务");

                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
```

```
            });
            try {
                Thread.sleep(sleep);
                sleep = sleep>100?sleep-100:sleep;
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }

    }
}
```

使用 java 自带线程池

ava 提供自带的线程池，而不需要自己去开发一个自定义线程池了。

线程池类 ThreadPoolExecutor 在包 java.util.concurrent 下

```
ThreadPoolExecutor threadPool= new ThreadPoolExecutor(10, 15, 60,
TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
```

第一个参数 10 表示这个线程池初始化了 10 个线程在里面工作
第二个参数 15 表示如果 10 个线程不够用了，就会自动增加到最多 15 个线程
第三个参数 60 结合第四个参数 TimeUnit.SECONDS，表示经过 60 秒，多出来的线程还没有接到活儿，就会回收，最后保持池子里就 10 个
第四个参数 TimeUnit.SECONDS 如上
第五个参数 new LinkedBlockingQueue() 用来放任务的集合

execute 方法用于添加新的任务

```
package multiplethread;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThread {

    public static void main(String[] args) throws
```

```
InterruptedException {

        ThreadPoolExecutor threadPool= new
ThreadPoolExecutor(10, 15, 60, TimeUnit.SECONDS, new
LinkedBlockingQueue<Runnable>());

        threadPool.execute(new Runnable(){

            @Override
            public void run() {
                // TODO Auto-generated method stub

                System.out.println("任务 1");

            }

        });

    }

}
```

练习- 借助线程池同步查找文件内容

在 练习-同步查找文件内容，如果文件特别多，就会创建很多的线程。 改写这个练习，使用线程池的方式来完成。

初始化一个大小是 10 的线程池

遍历所有文件，当遍历到文件是.java 的时候，创建一个查找文件的任务，把这个任务扔进线程池去执行，继续遍历下一个文件

```
package multiplethread;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class SearchFileTask implements Runnable{

    private File file;
    private String search;
    public SearchFileTask(File file,String search) {
        this.file = file;
        this.search= search;
    }
```

```
    public void run(){

        String fileContent = readFileConent(file);
        if(fileContent.contains(search)){
            System.out.printf( "线程: %s 找到子目标字符串%s,在
文件:%s%n",Thread.currentThread().getName(), search,file);

        }
    }

    public String readFileConent(File file){
        try (FileReader fr = new FileReader(file)) {
            char[] all = new char[(int) file.length()];
            fr.read(all);
            return new String(all);
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }

    }

}
package multiplethread;

import java.util.LinkedList;

public class ThreadPool {
    // 线程池大小
    int threadPoolSize;

    // 任务容器
    LinkedList<Runnable> tasks = new LinkedList<Runnable>();

    // 试图消费任务的线程


    public ThreadPool() {
```

```
        threadPoolSize = 10;

        // 启动10 个任务消费者线程
        synchronized (tasks) {
            for (int i = 0; i < threadPoolSize; i++) {
                new TaskConsumeThread("任务消费者线程 " +
i).start();
            }
        }
    }

    public void add(Runnable r) {
        synchronized (tasks) {
            tasks.add(r);
            // 唤醒等待的任务消费者线程
            tasks.notifyAll();
        }
    }

    class TaskConsumeThread extends Thread {
        public TaskConsumeThread(String name) {
            super(name);
        }

        Runnable task;

        public void run() {
            while (true) {
                synchronized (tasks) {
                    while (tasks.isEmpty()) {
                        try {
                            tasks.wait();
                        } catch (InterruptedException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                    }
                    task = tasks.removeLast();
                    // 允许添加任务的线程可以继续添加任务
                    tasks.notifyAll();
```

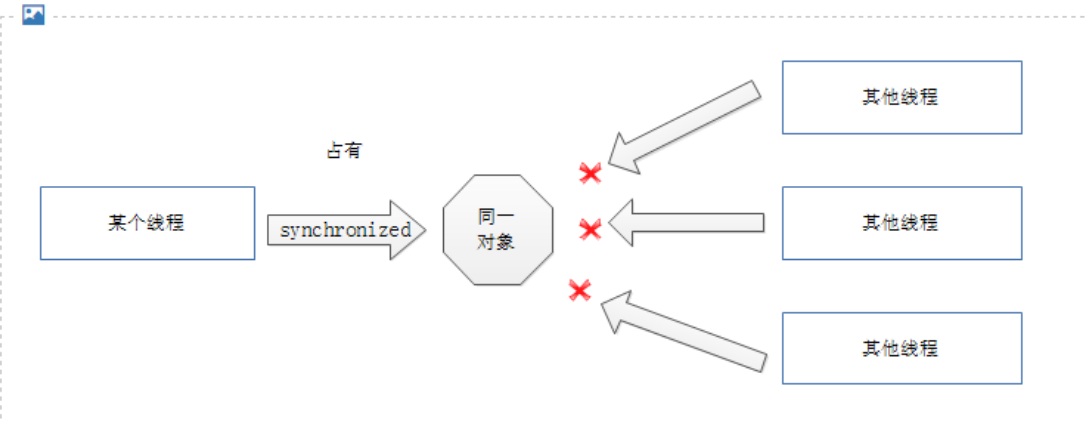
```
                }
                task.run();
            }
        }
    }

}
package multiplethread;

import java.io.File;

public class TestThread {

    static ThreadPool pool= new ThreadPool();
    public static void search(File file, String search) {

        if (file.isFile()) {

if(file.getName().toLowerCase().endsWith(".java")){
                SearchFileTask task = new
SearchFileTask(file, search);
                pool.add(task);
            }
        }
        if (file.isDirectory()) {
            File[] fs = file.listFiles();
            for (File f : fs) {
                search(f, search);
            }
        }
    }

    public static void main(String[] args) {
        File folder =new File("e:\\project");
        search(folder,"Magic");
    }
}
```

多线程 LOCK 对象

回忆 synchronized 同步的方式

首先回忆一下 synchronized 同步对象的方式

当一个线程占用 synchronized 同步对象，其他线程就不能占用了，直到释放这个同步对象为止




```
package multiplethread;

import java.text.SimpleDateFormat;
import java.util.Date;

public class TestThread {

    public static String now(){
        return new SimpleDateFormat("HH:mm:ss").format(new
Date());
    }

    public static void main(String[] args) {
        final Object someObject = new Object();

        Thread t1 = new Thread(){
            public void run(){
                try {

                    System.out.println( now()+" t1 线程已经运行
");

                    System.out.println( now()+this.getName()+
" 试图占有对象：someObject");
```

```
                    synchronized (someObject) {

System.out.println( now()+this.getName()+ " 占有对象：
someObject");
                        Thread.sleep(5000);

System.out.println( now()+this.getName()+ " 释放对象：
someObject");
                    }
                    System.out.println(now()+" t1 线程结束");
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };
        t1.setName(" t1");
        t1.start();
        Thread t2 = new Thread(){

            public void run(){
                try {
                    System.out.println( now()+" t2 线程已经运行
");
                    System.out.println( now()+this.getName()+
" 试图占有对象：someObject");
                    synchronized (someObject) {

System.out.println( now()+this.getName()+ " 占有对象：
someObject");
                        Thread.sleep(5000);

System.out.println( now()+this.getName()+ " 释放对象：
someObject");
                    }
                    System.out.println(now()+" t2 线程结束");
```

```
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        };
        t2.setName(" t2");
        t2.start();
    }

}
```

使用 Lock 对象实现同步效果

Lock 是一个接口，为了使用一个 Lock 对象，需要用到

```
Lock lock = new ReentrantLock();
```

与 synchronized (someObject) 类似的，lock()方法，表示当前线程占用 lock 对象，一旦占用，其他线程就不能占用了。
与 synchronized 不同的是，一旦 synchronized 块结束，就会自动释放对 someObject 的占用。 lock 却必须调用 unlock 方法进行手动释放，为了保证释放的执行，往往会把 unlock() 放在 finally 中进行。


```
package multiplethread;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

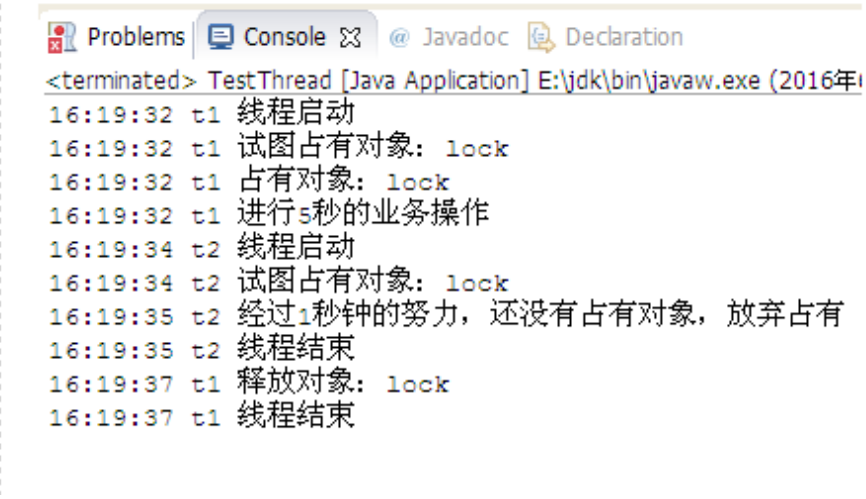
```
public class TestThread {

    public static String now() {
        return new SimpleDateFormat("HH:mm:ss").format(new
Date());
    }

    public static void log(String msg) {
        System.out.printf("%s %s %s %n", now() ,
Thread.currentThread().getName() , msg);
    }

    public static void main(String[] args) {
        Lock lock = new ReentrantLock();

        Thread t1 = new Thread() {
            public void run() {
                try {
                    log("线程启动");

                    log("试图占有对象：lock");


                    lock.lock();


                    log("占有对象：lock");

                    log("进行 5 秒的业务操作");
                    Thread.sleep(5000);

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    log("释放对象：lock");
                    lock.unlock();
                }
                log("线程结束");
            }
        };
        t1.setName("t1");
        t1.start();
```

```
        try {
            //先让 t1 飞 2 秒
            Thread.sleep(2000);
        } catch (InterruptedException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        Thread t2 = new Thread() {

            public void run() {
                try {
                    log("线程启动");

                    log("试图占有对象：lock");


                    lock.lock();


                    log("占有对象：lock");

                    log("进行 5 秒的业务操作");
                    Thread.sleep(5000);

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    log("释放对象：lock");
                    lock.unlock();
                }
                log("线程结束");
            }
        };
        t2.setName("t2");
        t2.start();
    }

}
```

trylock 方法

synchronized 是不占用到手不罢休的，会一直试图占用下去。
与 synchronized 的钻牛角尖不一样，Lock 接口还提供了一个 trylock 方法。
trylock 会在指定时间范围内试图占用，占成功了，就啪啪啪。 如果时间到了，还占用不成功，扭头就走~

注意：因为使用 trylock 有可能成功，有可能失败，所以后面 unlock 释放锁的时候，需要判断是否占用成功了，如果没占用成功也 unlock,就会抛出异常




```
package multiplethread;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class TestThread {

    public static String now() {
        return new SimpleDateFormat("HH:mm:ss").format(new
Date());
    }

    public static void log(String msg) {
        System.out.printf("%s %s %s %n", now() ,
Thread.currentThread().getName() , msg);
    }

    public static void main(String[] args) {
```

```
        Lock lock = new ReentrantLock();

        Thread t1 = new Thread() {
            public void run() {
                boolean locked = false;
                try {
                    log("线程启动");

                    log("试图占有对象：lock");


                    locked = lock.tryLock(1,TimeUnit.SECONDS);
                    if(locked){
                        log("占有对象：lock");

                        log("进行 5 秒的业务操作");

                        Thread.sleep(5000);
                    }
                    else{
                        log("经过 1 秒钟的努力，还没有占有对象，放
弃占有");

                    }

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {

                    if(locked){
                        log("释放对象：lock");

                        lock.unlock();
                    }
                }

                log("线程结束");

            }
        };
        t1.setName("t1");
        t1.start();
        try {

            //先让 t1 飞 2 秒
```

```
            Thread.sleep(2000);
        } catch (InterruptedException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        Thread t2 = new Thread() {

            public void run() {
                boolean locked = false;
                try {
                    log("线程启动");

                    log("试图占有对象：lock");


                    locked = lock.tryLock(1,TimeUnit.SECONDS);
                    if(locked){
                        log("占有对象：lock");

                        log("进行 5 秒的业务操作");
                        Thread.sleep(5000);
                    }
                    else{
                        log("经过 1 秒钟的努力，还没有占有对象，放
弃占有");

                    }

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {

                    if(locked){
                        log("释放对象：lock");
                        lock.unlock();
                    }
                }

                log("线程结束");

            }
        };
```

```
        t2.setName("t2");
        t2.start();
    }

}
```

线程交互

使用 synchronized 方式进行线程交互，用到的是同步对象的 wait,notify 和 notifyAll 方法

Lock 也提供了类似的解决办法，首先通过 lock 对象得到一个 Condition 对象，然后分别调用这个 Condition 对象的：**await**, **signal**,**signalAll** 方法

注意： 不是 Condition 对象的 wait,nofity,notifyAll 方法,是 await,signal,signalAll


```
package multiplethread;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class TestThread {

    public static String now() {
        return new SimpleDateFormat("HH:mm:ss").format(new
Date());
    }
```

```
    public static void log(String msg) {
        System.out.printf("%s %s %s %n", now() ,
Thread.currentThread().getName() , msg);
    }

    public static void main(String[] args) {
        Lock lock = new ReentrantLock();
        Condition condition = lock.newCondition();

        Thread t1 = new Thread() {
            public void run() {
                try {
                    log("线程启动");

                    log("试图占有对象：lock");


                    lock.lock();


                    log("占有对象：lock");

                    log("进行 5 秒的业务操作");
                    Thread.sleep(5000);
                    log("临时释放对象 lock，  并等待");
                    condition.await();
                    log("重新占有对象 lock，并进行 5 秒的业务操作
");
                    Thread.sleep(5000);

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    log("释放对象：lock");
                    lock.unlock();
                }
                log("线程结束");
            }
        };
```

```
        t1.setName("t1");
        t1.start();
        try {
            //先让 t1 飞 2 秒
            Thread.sleep(2000);
        } catch (InterruptedException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        Thread t2 = new Thread() {

            public void run() {
                try {
                    log("线程启动");

                    log("试图占有对象：lock");

                    lock.lock();

                    log("占有对象：lock");

                    log("进行 5 秒的业务操作");

                    Thread.sleep(5000);

                    log("唤醒等待中的线程");

                    condition.signal();

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    log("释放对象：lock");

                    lock.unlock();
                }

                log("线程结束");

            }
        };
        t2.setName("t2");
        t2.start();
    }
```

}

总结 Lock 和 synchronized 的区别

1. Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现，Lock 是代码层面的实现。

   2. Lock 可以选择性的获取锁，如果一段时间获取不到，可以放弃。synchronized 不行，会一根筋一直获取下去。 借助 Lock 的这个特性，就能够规避死锁，synchronized 必须通过谨慎和良好的设计，才能减少死锁的发生。

   3. synchronized 在发生异常和同步块结束的时候，会自动释放锁。而 Lock 必须手动释放， 所以如果忘记了释放锁，一样会造成死锁。

练习-借助 Lock，把 MyStack 修改为线程安全的类

在练习-线程安全的 MyStack 练习中，使用 synchronized 把 MyStack 修改为了线程安全的类。

接下来，借助 Lock 把 MyStack 修改为线程安全的类
把 synchronized 去掉
使用 lock 占用锁
使用 unlock 释放锁
必须放在 finally 执行，万一 heros.addLast 抛出异常也会执行

```
package multiplethread;

import java.util.LinkedList;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import charactor.Hero;

public class MyStack {

    LinkedList<Hero> heros = new LinkedList<Hero>();

    Lock lock = new ReentrantLock();

    //把 synchronized 去掉
    public  void push(Hero h) {
        try{
```

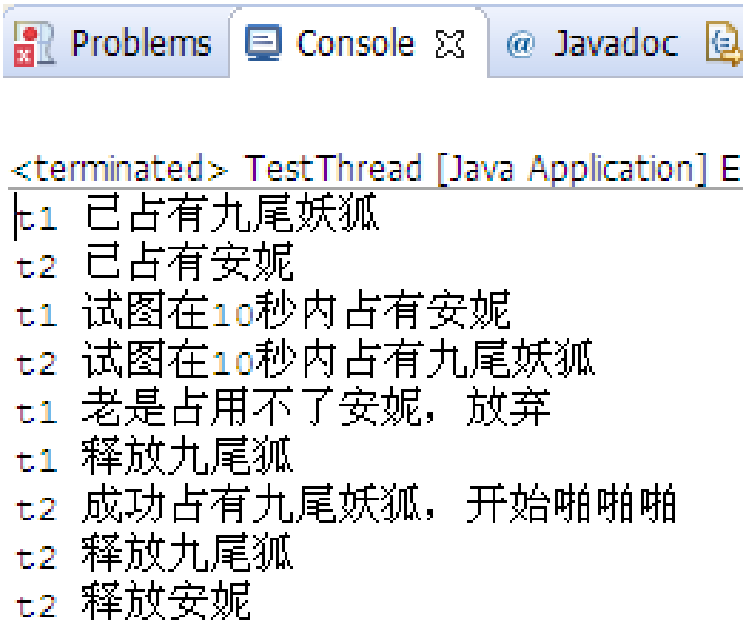
```
            //使用lock 占用锁
            lock.lock();
            heros.addLast(h);
        }
        finally{
            //使用unlock 释放锁

            //必须放在finally 执行，万一heros.addLast 抛出异常也
会执行
            lock.unlock();
        }

    }

    //把synchronized 去掉
    public  Hero pull() {
        try{
            //使用lock 占用锁
            lock.lock();
            return heros.removeLast();
        }
        finally{
            //使用unlock 释放锁

            //必须放在finally 执行，万一heros.removeLast();抛出
异常也会执行
            lock.unlock();
        }
    }

    public Hero peek() {
        return heros.getLast();
    }

    public static void main(String[] args) {

    }
```

}

练习-借助 tryLock 解决死锁问题

当多个线程按照不同顺序占用多个同步对象的时候，就有可能产生死锁现象。

死锁之所以会发生，就是因为 synchronized 如果占用不到同步对象，就会苦苦的一直等待下去，借助 tryLock 的有限等待时间，解决死锁问题




```
package multiplethread;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class TestThread {

    public static void main(String[] args) throws
InterruptedException {
        Lock lock_ahri = new ReentrantLock();
        Lock lock_annie = new ReentrantLock();

        Thread t1 = new Thread() {
            public void run() {

                // 占有九尾妖狐
```

```
                boolean ahriLocked = false;
                boolean annieLocked = false;

                try {
                    ahriLocked = lock_ahri.tryLock(10,
TimeUnit.SECONDS);
                    if (ahriLocked) {
                        System.out.println("t1 已占有九尾妖狐
");
                        // 停顿1000秒，另一个线程有足够的时间占
有安妮
                        Thread.sleep(1000);
                        System.out.println("t1 试图在10秒内占
有安妮");
                        try {
                            annieLocked =
lock_annie.tryLock(10, TimeUnit.SECONDS);
                            if (annieLocked)
                                System.out.println("t1 成功占有
安妮，开始啪啪啪");
                            else{
                                System.out.println("t1 老是占用
不了安妮，放弃");
                            }
                        } finally {
                            if (annieLocked){
                                System.out.println("t1 释放安妮
");
                                lock_annie.unlock();
                            }
                        }

                    }
```

```
                } catch (InterruptedException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                } finally {
                    if (ahriLocked){
                        System.out.println("t1 释放九尾狐");
                        lock_ahri.unlock();
                    }
                }

            }
        };
        t1.start();

        Thread.sleep(100);

        Thread t2 = new Thread() {
            public void run() {
                boolean annieLocked = false;
                boolean ahriLocked = false;

                try {annieLocked = lock_annie.tryLock(10,
TimeUnit.SECONDS);

                    if (annieLocked){

                        System.out.println("t2 已占有安妮");

                        // 停顿1000秒，另一个线程有足够的时间占
有安妮

                        Thread.sleep(1000);

                        System.out.println("t2 试图在10秒内占
有九尾妖狐");

                        try {
                            ahriLocked = lock_ahri.tryLock(10,
TimeUnit.SECONDS);

                            if (ahriLocked)

                                System.out.println("t2 成功占有
```

```
九尾妖狐，开始啪啪啪");
                            else{
                                System.out.println("t2 老是占用
不了九尾妖狐，放弃");
                            }
                        }
                        finally {
                            if (ahriLocked){
                                System.out.println("t2 释放九尾
狐");
                                lock_ahri.unlock();
                            }

                        }

                    }
                } catch (InterruptedException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                } finally {
                    if (annieLocked){
                        System.out.println("t2 释放安妮");
                        lock_annie.unlock();
                    }

                }
            }
        };
        t2.start();

    }
}
```

练习-生产者消费者问题

在练习-生产者消费者问题这个练习中，是用 wait(), nofity(), notifyAll 实现了。

接下来使用 Condition 对象的：**await**, **signal**,**signalAll** 方法实现同样的效果

```java
package multiplethread;

import java.util.LinkedList;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class MyStack<T> {

    LinkedList<T> values = new LinkedList<T>();

    Lock lock = new ReentrantLock();
    Condition condition = lock.newCondition();

    public void push(T t) {
        try {
            lock.lock();
            while (values.size() >= 200) {
                try {
                    condition.await();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            condition.signalAll();
            values.addLast(t);
        } finally {
            lock.unlock();
        }

    }

    public T pull() {
        T t=null;
        try {
            lock.lock();
            while (values.isEmpty()) {
                try {
                    condition.await();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
```

```
            }
            condition.signalAll();
            t= values.removeLast();
        } finally {
            lock.unlock();
        }
        return t;
    }

    public T peek() {
        return values.getLast();
    }
}
```

多线程 原子访问

原子性操作概念

所谓的**原子性操作**即不可中断的操作，比如赋值操作

```
<br data-filtered="filtered">int i = 5;<br data-filtered="filtered">
```

原子性操作本身是线程安全的
但是 i++ 这个行为，事实上是有 3 个原子性操作组成的。
步骤 1. 取 i 的值
步骤 2. i + 1
步骤 3. 把新的值赋予 i
这三个步骤，每一步都是一个原子操作，但是合在一起，就不是原子操作。就**不是线程安全**的。
换句话说，一个线程在步骤 1 取 i 的值结束后，还没有来得及进行步骤 2，另一个线程也可以取 i 的值了。
这也是分析同步问题产生的原因 中的原理。
i++ ， i--， i = i+1 这些都是非原子性操作。
只有 int i = 1,这个赋值操作是原子性的。

AtomicInteger

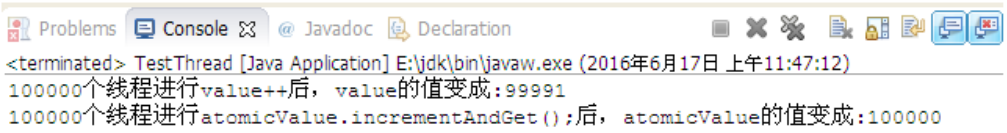
JDK6 以后，新增加了一个包 **java.util.concurrent.atomic**，里面有各种原子类，比如 **AtomicInteger**。
而 AtomicInteger 提供了各种自增，自减等方法，这些方法都是原子性的。 换句话说，自增方法 **incrementAndGet** 是线程安全的，同一个时间，只有一个线程可以

调用这个方法。

```
package multiplethread;

import java.util.concurrent.atomic.AtomicInteger;

public class TestThread {

        public static void main(String[] args) throws
InterruptedException {
                AtomicInteger atomicI =new AtomicInteger();
                int i = atomicI.decrementAndGet();
                int j = atomicI.incrementAndGet();
                int k = atomicI.addAndGet(3);

        }

}
```

同步测试

分别使用基本变量的非原子性的++运算符和 原子性的 AtomicInteger 对象的 incrementAndGet 来进行多线程测试。
测试结果如图所示




```
package multiplethread;

import java.util.concurrent.atomic.AtomicInteger;

public class TestThread {

    private static int value = 0;
    private static AtomicInteger atomicValue =new
AtomicInteger();
    public static void main(String[] args) {
        int number = 100000;
        Thread[] ts1 = new Thread[number];
        for (int i = 0; i < number; i++) {
            Thread t =new Thread(){
                public void run(){
                    value++;
```

```
            }
        };
        t.start();
        ts1[i] = t;
    }

    //等待这些线程全部结束
    for (Thread t : ts1) {
        try {
            t.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    System.out.printf("%d 个线程进行 value++后，value 的值变
成:%d%n", number,value);
    Thread[] ts2 = new Thread[number];
    for (int i = 0; i < number; i++) {
        Thread t =new Thread(){
            public void run(){
                atomicValue.incrementAndGet();
            }
        };
        t.start();
        ts2[i] = t;
    }

    //等待这些线程全部结束
    for (Thread t : ts2) {
        try {
            t.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    System.out.printf("%d 个线程进行
```

```
atomicValue.incrementAndGet();后，atomicValue 的值变成:%d%n",
number,atomicValue.intValue());
    }

}
```

练习-使用 AtomicInteger 来替换 Hero 类中的 synchronized

在给 Hero 的方法加上修饰符 synchronized 这个知识点中，通过给 hurt 和 recover 方法加上 synchronized 来达到线程安全的效果。

这一次换成使用 AtomicInteger 来解决这个问题
把 Hero 的 hp 设计为

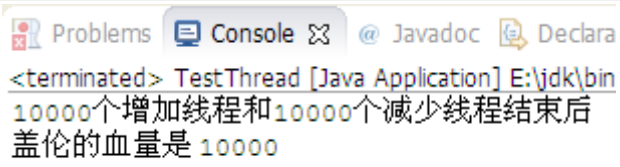
```
AtomicInteger hp = new AtomicInteger();
```

recover 和 hurt 上的 synchronized 修饰符去掉
recover 中调用

```
hp.incrementAndGet();
```

hurt 中调用

```
hp.decrementAndGet();
```




```
package charactor;

import java.util.concurrent.atomic.AtomicInteger;

public class Hero{
    public String name;
    public AtomicInteger hp = new AtomicInteger();
```

```
    public int damage;

    public  void recover(){
        hp.incrementAndGet();
    }

    public void hurt(){
        hp.decrementAndGet();
    }

    public void attackHero(Hero h) {
        h.hp.addAndGet(0-damage);
        System.out.format("%s 正在攻击 %s, %s 的血变成了 %.0f%n",name,h.name,h.name,h.hp);
        if(h.isDead())
            System.out.println(h.name +"死了！");
    }

    public boolean isDead() {
        return 0>=hp.intValue()?true:false;
    }

}
package multiplethread;

import java.lang.reflect.GenericSignatureFormatError;

import charactor.Hero;

public class TestThread {

        public static void main(String[] args) {

                final Hero gareen = new Hero();
                gareen.name = "盖伦";
                gareen.hp.set(10000);
                int n = 10000;

                Thread[] addThreads = new Thread[n];
                Thread[] reduceThreads = new Thread[n];
```

```
for (int i = 0; i < n; i++) {
        Thread t = new Thread(){
                public void run(){

                        //recover 自带 synchronized
                        gareen.recover();

                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        };
        t.start();
        addThreads[i] = t;

}

for (int i = 0; i < n; i++) {
        Thread t = new Thread(){
                public void run(){
                        //hurt 自带 synchronized
                        gareen.hurt();

                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        };
        t.start();
        reduceThreads[i] = t;
}

for (Thread t : addThreads) {
        try {
                t.join();
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

```
                }
            }
            for (Thread t : reduceThreads) {
                try {
                    t.join();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

            System.out.printf("%d 个增加线程和%d 个减少线程结束后%n 盖伦的血量是 %

        }

}
```

双击选中所有代码

常见的线程安全相关的面试题

HashMap 和 Hashtable 的区别

HashMap 和 Hashtable 都实现了 Map 接口，都是键值对保存数据的方式
区别 1：
HashMap 可以存放 null
Hashtable 不能存放 null
区别 2：
HashMap 不是线程安全的类
Hashtable 是线程安全的类

StringBuffer 和 StringBuilder 的区别

StringBuffer 是线程安全的
StringBuilder 是非线程安全的

所以当进行大量字符串拼接操作的时候，如果是单线程就用 StringBuilder 会更快些，如果是多线程，就需要用 StringBuffer 保证数据的安全性

非线程安全的为什么会比线程安全的 快？ 因为必须要同步嘛，省略了些时间

ArrayList 和 Vector 的区别

ArrayList 类的声明：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Vector 类的声明：

```
public class Vector<E>      extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

一模一样的~
他们的区别也在于，Vector 是线程安全的类，而 ArrayList 是非线程安全的。

把非线程安全的集合转换为线程安全

ArrayList 是非线程安全的，换句话说，多个线程可以同时进入**一个 ArrayList 对象**的 add 方法

借助 Collections.synchronizedList，可以把 ArrayList 转换为线程安全的 List。

与此类似的，还有 HashSet,LinkedList,HashMap 等等非线程安全的类，都通过工具类 Collections 转换为线程安全的

```
package multiplethread;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestThread {

    public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 =
Collections.synchronizedList(list1);
    }

}
```

练习-线程安全的 MyStack

借助把非线程安全的集合转换为线程安全，用另一个方式完成 练习-线程安全的 MyStack
把 LinkedList 通过 Collections.synchronizedList 转换成了一个线程安全的 List

```
List<Hero> heros = (List<Hero>) Collections.synchronizedList(new LinkedList<Hero>());
```

不需要在 push 上加 synchronized 修饰符
虽然多个线程可以同时进入 push 方法，但是调用 heros.add 方法的时候
同一时间，只有一个线程可以进入

```
public void push(Hero h) {
   heros.add(h);
}
```

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

import charactor.Hero;

public class MyStack implements Stack{

    //把 LinkedList 通过 Collections.synchronizedList 转换成了一个线程安全的 List
    List<Hero> heros = (List<Hero>) Collections.synchronizedList(new LinkedList<Hero>());

    //不需要在 push 上加 synchronized 修饰符
    //虽然多个线程可以同时进入 push 方法，但是调用 heros.add 方法的时候
    //同一时间，只有一个线程可以进入
    public void push(Hero h) {
```

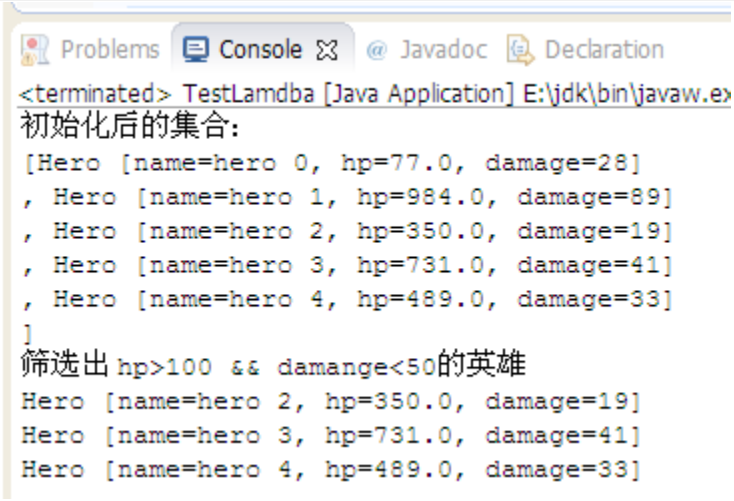
```
        heros.add(h);
    }

    public Hero pull() {
        return heros.remove(heros.size()-1);
    }

    public Hero peek() {
        return heros.get(heros.size()-1);
    }
}
```

LAMBDA 表达式

普通方法

使用一个普通方法，在 for 循环遍历中进行条件判断，筛选出满足条件的数据

```
hp>100 && damage<50
```




```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLambda {
    public static void main(String[] args) {
```

```
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 10; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合: ");

        System.out.println(heros);

        System.out.println("筛选出 hp>100 && damange<50 的英雄
");
        filter(heros);
    }

    private static void filter(List<Hero> heros) {
        for (Hero hero : heros) {
            if(hero.hp>100 && hero.damage<50)
                System.out.print(hero);
        }
    }

}
package charactor;

public class Hero implements Comparable<Hero>{
    public String name;
    public float hp;

    public int damage;

    public Hero(){

    }

    public Hero(String name) {
        this.name =name;

    }


    //初始化 name,hp,damage 的构造方法
    public Hero(String name,float hp, int damage) {
        this.name =name;
```

```
        this.hp = hp;
        this.damage = damage;
    }

    @Override
    public int compareTo(Hero anotherHero) {
        if(damage<anotherHero.damage)
            return 1;
        else
            return -1;
    }

    @Override
    public String toString() {
        return "Hero [name=" + name + ", hp=" + hp + ",
damage=" + damage + "]\r\n";
    }

}
```

匿名类方式

首先准备一个接口 HeroChecker，提供一个 test(Hero)方法
然后通过匿名类的方式，实现这个接口

```
HeroChecker checker = new HeroChecker() {
    public boolean test(Hero h) {
        return (h.hp>100 && h.damage<50);
    }
};
```

接着调用 filter，传递这个 checker 进去进行判断，这种方式就很像通过 Collections.sort 在对一个 Hero 集合排序，需要传一个 Comparator 的匿名类对象进去一样。


```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLambda {
    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);

        System.out.println("使用匿名类的方式，筛选出 hp>100 &&

damange<50 的英雄");

        HeroChecker checker = new HeroChecker() {
            @Override
            public boolean test(Hero h) {
                return (h.hp>100 && h.damage<50);
            }
        };

        filter(heros,checker);
```

```
    }

    private static void filter(List<Hero> heros,HeroChecker
checker) {
        for (Hero hero : heros) {
            if(checker.test(hero))
                System.out.print(hero);
        }
    }

}
package lambda;

import charactor.Hero;

public interface HeroChecker {
    public boolean test(Hero h);
}
```

Lambda 方式

使用 Lambda 方式筛选出数据

```
filter(heros,(h)->h.hp>100 && h.damage<50);
```

同样是调用 filter 方法，从上一步的传递匿名类对象，变成了传递一个 Lambda 表达式进去

```
h->h.hp>100 && h.damage<50
```

咋一看 Lambda 表达式似乎不好理解，其实很简单，下一步讲解如何从一个匿名类一点点演变成 Lambda 表达式

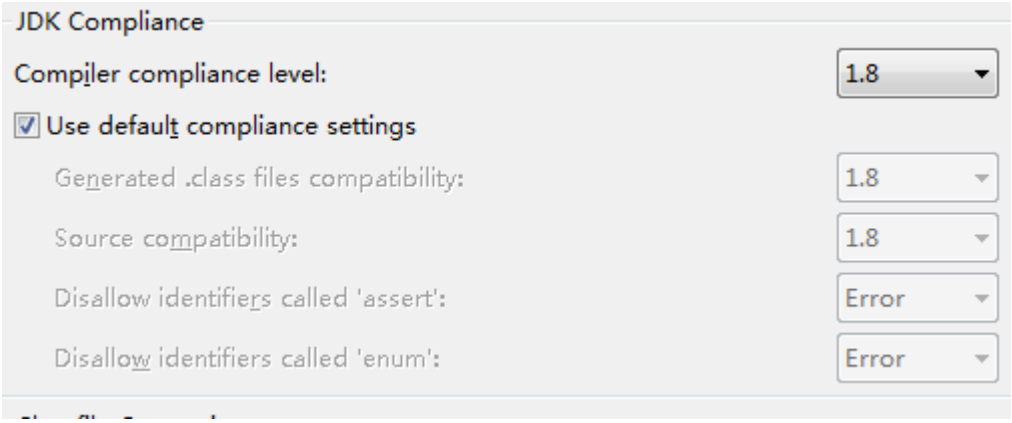

```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLamdba {
    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);

        System.out.println("使用 Lamdba 的方式，筛选出 hp>100

&& damange<50 的英雄");

        filter(heros,h->h.hp>100 && h.damage<50);
    }

    private static void filter(List<Hero> heros,HeroChecker
checker) {
        for (Hero hero : heros) {
            if(checker.test(hero))
                System.out.print(hero);
        }
    }
```

}

设置 eclipse 以支持 Lambda

如果你的 eclipse 能够正常识别 Lambda，那么就可以跳过这个章节了。
因为 Lambda 是 JDK8 的内容，除了 JDK 需要使用 1.8 以上版本外(在 JDK 环境变量配置下载的就是 1.8 了)，还需要在 eclipse 中把编译器设置为 1.8 才能够正常识别 Lambda.

设置办法：
菜单->Window->Preferences->Java-Compiler->Compiler compliance leve: 设置为 1.8 即可



从匿名类演变成 Lambda 表达式

Lambda 表达式可以看成是匿名类一点点演变过来
1. 匿名类的正常写法

```
HeroChecker c1 = new HeroChecker() {
        public boolean test(Hero h) {
                return (h.hp>100 && h.damage<50);
        }
};
```

2. 把外面的壳子去掉
只保留方法参数和方法体
参数和方法体之间加上符号 ->

```
HeroChecker c2 = (Hero h) ->{
        return h.hp>100 && h.damage<50;
```

```
};
```

3. 把 return 和{}去掉

```
HeroChecker c3 = (Hero h) ->h.hp>100 && h.damage<50;
```

4. 把 参数类型和圆括号去掉(只有一个参数的时候，才可以去掉圆括号)

```
HeroChecker c4 = h ->h.hp>100 && h.damage<50;
```

5. 把 c4 作为参数传递进去

```
filter(heros,c4);
```

6. 直接把表达式传递进去

```
filter(heros, h -> h.hp > 100 && h.damage < 50);
```

```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLamdba {
    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合: ");

        System.out.println(heros);
```

```
        System.out.println("使用匿名类的方式，筛选出 hp>100 &&
damange<50 的英雄");

        // 匿名类的正常写法
        HeroChecker c1 = new HeroChecker() {
            @Override
            public boolean test(Hero h) {
                return (h.hp > 100 && h.damage < 50);
            }
        };
        // 把new HeroChcekcer，方法名，方法返回类型信息去掉

        // 只保留方法参数和方法体

        // 参数和方法体之间加上符号 ->
        HeroChecker c2 = (Hero h) -> {
            return h.hp > 100 && h.damage < 50;
        };

        // 把return 和{}去掉
        HeroChecker c3 = (Hero h) -> h.hp > 100 && h.damage <
50;

        // 把 参数类型和圆括号去掉
        HeroChecker c4 = h -> h.hp > 100 && h.damage < 50;

        // 把c4 作为参数传递进去
        filter(heros, c4);

        // 直接把表达式传递进去
        filter(heros, h -> h.hp > 100 && h.damage < 50);
    }

    private static void filter(List<Hero> heros, HeroChecker
checker) {
        for (Hero hero : heros) {
            if (checker.test(hero))
```

```
                System.out.print(hero);
            }
        }

}
```

匿名方法

与匿名类 概念相比较，
Lambda 其实就是**匿名方法**，这是一种**把方法作为参数**进行传递的编程思想。

虽然代码是这么写

```
filter(heros, h -> h.hp > 100 && h.damage < 50);
```

但是，Java 会在背后，悄悄的，把这些都还原成匿名类方式。
引入 Lambda 表达式，会使得代码更加紧凑，而不是各种接口和匿名类到处飞。

Lambda 的弊端

Lambda 表达式虽然带来了代码的简洁，但是也有其局限性。
1. 可读性差，与**啰嗦的**但是**清晰的**匿名类代码结构比较起来，Lambda 表达式一旦变得比较长，就难以理解
2. 不便于调试，很难在 Lambda 表达式中增加调试信息，比如日志
3. 版本支持，Lambda 表达式在 JDK8 版本中才开始支持，如果系统使用的是以前的版本，考虑系统的稳定性等原因，而不愿意升级，那么就无法使用。

Lambda 比较适合用在简短的业务代码中，并不适合用在复杂的系统中，会加大维护成本。

练习-Comparator

把比较器-Comparator 章节中的代码，按照从匿名类演变成 Lambda 表达式的步骤，改写为 Lambda 表达式

```
package collection;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;
```

```
import charactor.Hero;

public class TestCollection {
    public static void main(String[] args) {
        Random r =new Random();
        List<Hero> heros = new ArrayList<Hero>();

        for (int i = 0; i < 10; i++) {
            heros.add(new Hero("hero "+ i, r.nextInt(100),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);


        //匿名类

        Comparator<Hero> c = new Comparator<Hero>() {
            @Override
            public int compare(Hero h1, Hero h2) {
                return h1.hp>=h2.hp?1:-1;
            }
        };


        //Lambda 表达式

        c = (Hero h1, Hero h2)-> {
            return h1.hp>=h2.hp?1:-1;
        };

        //去掉 return 和大括号

        c = (Hero h1, Hero h2)->h1.hp>=h2.hp?1:-1;

        //去掉 参数类型

        c = (h1, h2)->h1.hp>=h2.hp?1:-1;

        //有两个参数，无法去掉圆括号

        Collections.sort(heros,c);


        //直接把 Lambda 表达式作为参数传进去

        Collections.sort(heros,(h1, h2)->h1.hp>=h2.hp?1:-1);
        System.out.println(heros);
```

```
    }
}
```

LAMBDA 方法引用

引用静态方法

首先为 TestLambda 添加一个静态方法：

```
public static boolean testHero(Hero h) {
    return h.hp>100 && h.damage<50;
}
```

Lambda 表达式：

```
filter(heros, h->h.hp>100 && h.damage<50);
```

在 Lambda 表达式中调用这个静态方法：

```
filter(heros, h -> TestLambda.testHero(h) );
```

调用静态方法还可以改写为：

```
filter(heros, TestLambda::testHero);
```

这种方式就叫做**引用静态方法**

```
<terminated> TestLamdba [Java Application] D:\jdk\
初始化后的集合：
[Hero [name=hero 0, hp=801.0, damage=16]
, Hero [name=hero 1, hp=101.0, damage=19]
, Hero [name=hero 2, hp=72.0, damage=73]
, Hero [name=hero 3, hp=696.0, damage=64]
, Hero [name=hero 4, hp=552.0, damage=75]
]
使用匿名类过滤
Hero [name=hero 0, hp=801.0, damage=16]
Hero [name=hero 1, hp=101.0, damage=19]
使用Lambda表达式
Hero [name=hero 0, hp=801.0, damage=16]
Hero [name=hero 1, hp=101.0, damage=19]
在Lambda表达式中使用静态方法
Hero [name=hero 0, hp=801.0, damage=16]
Hero [name=hero 1, hp=101.0, damage=19]
直接引用静态方法
Hero [name=hero 0, hp=801.0, damage=16]
Hero [name=hero 1, hp=101.0, damage=19]
```

```java
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLambda {
    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);

        HeroChecker c = new HeroChecker() {
            public boolean test(Hero h) {
                return h.hp>100 && h.damage<50;
            }
        };

        System.out.println("使用匿名类过滤");
```

```
        filter(heros, c);
        System.out.println("使用 Lambda 表达式");
        filter(heros, h->h.hp>100 && h.damage<50);
        System.out.println("在 Lambda 表达式中使用静态方法");
        filter(heros, h -> TestLambda.testHero(h) );
        System.out.println("直接引用静态方法");
        filter(heros, TestLambda::testHero);
    }

    public static boolean testHero(Hero h) {
        return h.hp>100 && h.damage<50;
    }

    private static void filter(List<Hero> heros, HeroChecker
checker) {
        for (Hero hero : heros) {
            if (checker.test(hero))
                System.out.print(hero);
        }
    }

}
```

引用对象方法

与引用静态方法很类似，只是传递方法的时候，需要一个对象的存在

```
TestLambda testLambda = new TestLambda();
filter(heros, testLambda::testHero);
```

这种方式叫做引用对象方法


```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLambda {
    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);


        System.out.println("使用引用对象方法  的过滤结果：");

        //使用类的对象方法

        TestLambda testLambda = new TestLambda();
        filter(heros, testLambda::testHero);
    }

    public boolean testHero(Hero h) {
        return h.hp>100 && h.damage<50;
    }
```

```
    private static void filter(List<Hero> heros, HeroChecker
checker) {
        for (Hero hero : heros) {
            if (checker.test(hero))
                System.out.print(hero);
        }
    }

}
```

引用容器中的对象的方法

首先为 Hero 添加一个方法

```
public boolean matched(){
    return this.hp>100 && this.damage<50;
}
```

使用 Lambda 表达式

```
filter(heros,h-> h.hp>100 && h.damage<50 );
```

在 Lambda 表达式中调用容器中的对象 Hero 的方法 matched

```
filter(heros,h-> h.matched() );
```

matched 恰好就是容器中的对象 Hero 的方法，那就可以进一步改写为

```
filter(heros, Hero::matched);
```

这种方式就叫做**引用容器中的对象的方法**

```
<terminated> TestLamdba [Java Application] E:\jdk\bin\javaw.
初始化后的集合:
[Hero [name=hero 0, hp=313.0, damage=78]
, Hero [name=hero 1, hp=288.0, damage=47]
, Hero [name=hero 2, hp=976.0, damage=54]
, Hero [name=hero 3, hp=763.0, damage=87]
, Hero [name=hero 4, hp=659.0, damage=7]
]
Lambda表达式:
Hero [name=hero 1, hp=288.0, damage=47]
Hero [name=hero 4, hp=659.0, damage=7]
Lambda表达式中调用容器中的对象的matched方法:
Hero [name=hero 1, hp=288.0, damage=47]
Hero [name=hero 4, hp=659.0, damage=7]
引用容器中对象的方法 之过滤结果:
Hero [name=hero 1, hp=288.0, damage=47]
Hero [name=hero 4, hp=659.0, damage=7]
```

```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestLambda {
    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);


        System.out.println("Lambda 表达式：");

        filter(heros,h-> h.hp>100 && h.damage<50 );


        System.out.println("Lambda 表达式中调用容器中的对象的

matched 方法：");

        filter(heros,h-> h.matched() );
```

```
        System.out.println("引用容器中对象的方法 之过滤结果：");
        filter(heros, Hero::matched);
    }

    public boolean testHero(Hero h) {
        return h.hp>100 && h.damage<50;
    }

    private static void filter(List<Hero> heros, HeroChecker
checker) {
        for (Hero hero : heros) {
            if (checker.test(hero))
                System.out.print(hero);
        }
    }

}
```

引用构造器

有的接口中的方法会返回一个对象，比如 java.util.function.Supplier 提供了一个 get 方法，返回一个对象。

```
public interface Supplier<T> {
    T get();
}
```

设计一个方法，参数是这个接口

```
public static List getList(Supplier<List> s){
  return s.get();
}
```

为了调用这个方法，有 3 种方式
第一种匿名类：

```
Supplier<List> s = new Supplier<List>() {
    public List get() {
        return new ArrayList();
    }
```

```
};
List list1 = getList(s);
```

第二种：Lambda 表达式

```
List list2 = getList(()->new ArrayList());
```

第三种：引用构造器

```
List list3 = getList(ArrayList::new);
```

```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Supplier;

public class TestLambda {
    public static void main(String[] args) {
        Supplier<List> s = new Supplier<List>() {
            public List get() {
                return new ArrayList();
            }
        };

        //匿名类
        List list1 = getList(s);

        //Lambda 表达式
        List list2 = getList(()->new ArrayList());

        //引用构造器
        List list3 = getList(ArrayList::new);

    }

    public static List getList(Supplier<List> s){
        return s.get();
```

```
    }

}
```

练习-引用静态方法

把比较器-Comparator 章节中的代码，使用引用静态方法的方式来实现

```
package collection;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestCollection {
    public static void main(String[] args) {
        Random r =new Random();
        List<Hero> heros = new ArrayList<Hero>();

        for (int i = 0; i < 10; i++) {
            heros.add(new Hero("hero "+ i, r.nextInt(100),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);


Collections.sort(heros,(h1,h2)->TestCollection.compare(h1,
h2));

        Collections.sort(heros,TestCollection::compare);

        System.out.println(heros);
    }

    public static int compare(Hero h1, Hero h2){
        return h1.hp>=h2.hp?1:-1;
    }
}
```

练习-引用容器中的对象的方法

把比较器-Comparator 章节中的代码，使用 引用容器中的对象的方法 的方式来实现。

提示：为 Hero 提供一个 compareHero 方法

```
package charactor;

public class Hero{
    public String name;
    public float hp;

    public int damage;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getHp() {
        return hp;
    }
    public void setHp(float hp) {
        this.hp = hp;
    }
    public int getDamage() {
        return damage;
    }
    public void setDamage(int damage) {
        this.damage = damage;
    }
    public Hero(){

    }
    public Hero(String name) {
        this.name =name;
    }

    public boolean matched(){
        return this.hp>100 && this.damage<50;
    }

    public Hero(String name,float hp, int damage) {
```

```
        this.name =name;
        this.hp = hp;
        this.damage = damage;
    }

    public int compareHero(Hero h){
        return hp>=h.hp?-1:1;
    }

    public String toString() {
        return "Hero [name=" + name + ", hp=" + hp + ",
damage=" + damage + "]\r\n";
    }

}
package collection;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestCollection {
    public static void main(String[] args) {
        Random r =new Random();
        List<Hero> heros = new ArrayList<Hero>();

        for (int i = 0; i < 10; i++) {
            heros.add(new Hero("hero "+ i, r.nextInt(100),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);

        Comparator<Hero> c = (h1,h2)-> h1.compareHero(h2) ;
        Collections.sort(heros,c);

        Collections.sort(heros,Hero::compareHero);

        System.out.println(heros);
```

```
    }

}
```

练习-引用构造器

把比较 ArrayList 和 LinkedList 的区别这段代码，改造成引用构造器的模式。
目前的调用方式是:

```
        List<Integer> l;
        l = new ArrayList<>();
        insertFirst(l, "ArrayList");

        l = new LinkedList<>();
        insertFirst(l, "LinkedList");
```

改造后的调用方式将变为：

```
        insertFirst(ArrayList::new, "ArrayList");
        insertFirst(LinkedList::new, "LinkedList");
```

```
package collection;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class TestCollection {
    public static void main(String[] args) {
        List<Integer> l;
        l = new ArrayList<>();
        insertFirst(l, "ArrayList");

        l = new LinkedList<>();
        insertFirst(l, "LinkedList");

    }

    private static void insertFirst(List<Integer> l, String
type) {
        int total = 1000 * 100;
        final int number = 5;
        long start = System.currentTimeMillis();
```

```
        for (int i = 0; i < total; i++) {
            l.add(0, number);
        }
        long end = System.currentTimeMillis();
        System.out.printf("在%s 最前面插入%d 条数据，总共耗时 %d
毫秒 %n", type, total, end - start);
    }

}
```

答案-引用构造器

```
package collection;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Supplier;

public class TestCollection {
    public static void main(String[] args) {
        insertFirst(ArrayList::new, "ArrayList");
        insertFirst(LinkedList::new, "LinkedList");
    }

    private static void insertFirst(Supplier<List> s, String
type) {
        int total = 1000 * 100;
        final int number = 5;
        long start = System.currentTimeMillis();
        List list = s.get();
        for (int i = 0; i < total; i++) {
            list.add(0, number);
        }
        long end = System.currentTimeMillis();
        System.out.printf("在%s 最前面插入%d 条数据，总共耗时 %d
毫秒 %n", type, total, end - start);
    }

}
```

AVA 集合的聚合操作

传统方式与聚合操作方式遍历数据

遍历数据的传统方式就是使用 for 循环，然后条件判断，最后打印出满足条件的数据

```
for (Hero h : heros) {
    if (h.hp > 100 && h.damage < 50)
        System.out.println(h.name);
}
```

使用聚合操作方式，画风就发生了变化：

```
heros
    .stream()
    .filter(h -> h.hp > 100 && h.damage < 50)
    .forEach(h -> System.out.println(h.name));
```

```
<terminated> TestAggregate [Java Application] E:\jdk\bin\jav
初始化后的集合:
[Hero [name=hero 0, hp=199.0, damage=30]
, Hero [name=hero 1, hp=158.0, damage=49]
, Hero [name=hero 2, hp=390.0, damage=64]
, Hero [name=hero 3, hp=284.0, damage=21]
, Hero [name=hero 4, hp=950.0, damage=78]
]
查询条件: hp>100 && damage<50
通过传统操作方式找出满足条件的数据:
hero 0
hero 1
hero 3
通过聚合操作方式找出满足条件的数据:
hero 0
hero 1
hero 3
```

```
package lambda;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import charactor.Hero;
```

```
public class TestAggregate {

    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化后的集合：");

        System.out.println(heros);

        System.out.println("查询条件：hp>100 && damage<50");

        System.out.println("通过传统操作方式找出满足条件的数据：
");

        for (Hero h : heros) {
            if (h.hp > 100 && h.damage < 50)
                System.out.println(h.name);
        }

        System.out.println("通过聚合操作方式找出满足条件的数据：
");
        heros
                .stream()
                .filter(h -> h.hp > 100 && h.damage < 50)
                .forEach(h -> System.out.println(h.name));

    }
}
```

Stream 和管道的概念

```
heros
    .stream()
    .filter(h -> h.hp > 100 && h.damage < 50)
    .forEach(h -> System.out.println(h.name));
```

要了解聚合操作，首先要建立 Stream 和管道的概念

Stream 和 Collection 结构化的数据不一样，Stream 是一系列的元素，就像是生产线上的罐头一样，一串串的出来。
管道指的是一系列的聚合操作。

管道又分 3 个部分
管道源：在这个例子里，源是一个 List
中间操作： 每个中间操作，又会返回一个 Stream，比如.filter()又返回一个 Stream，中间操作是“懒”操作，并不会真正进行遍历。
结束操作：当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。 结束操作不会返回 Stream，但是会返回 int、float、String、Collection 或者像 forEach，什么都不返回，结束操作才进行真正的遍历行为，在遍历的时候，才会去进行中间操作的相关判断

注： 这个 Stream 和 I/O 章节的 InputStream,OutputStream 是不一样的概念。

管道源

把 Collection 切换成管道源很简单，调用 stream()就行了。

```
heros.stream()
```

但是数组却没有 stream()方法，需要使用

```
Arrays.stream(hs)
```

或者

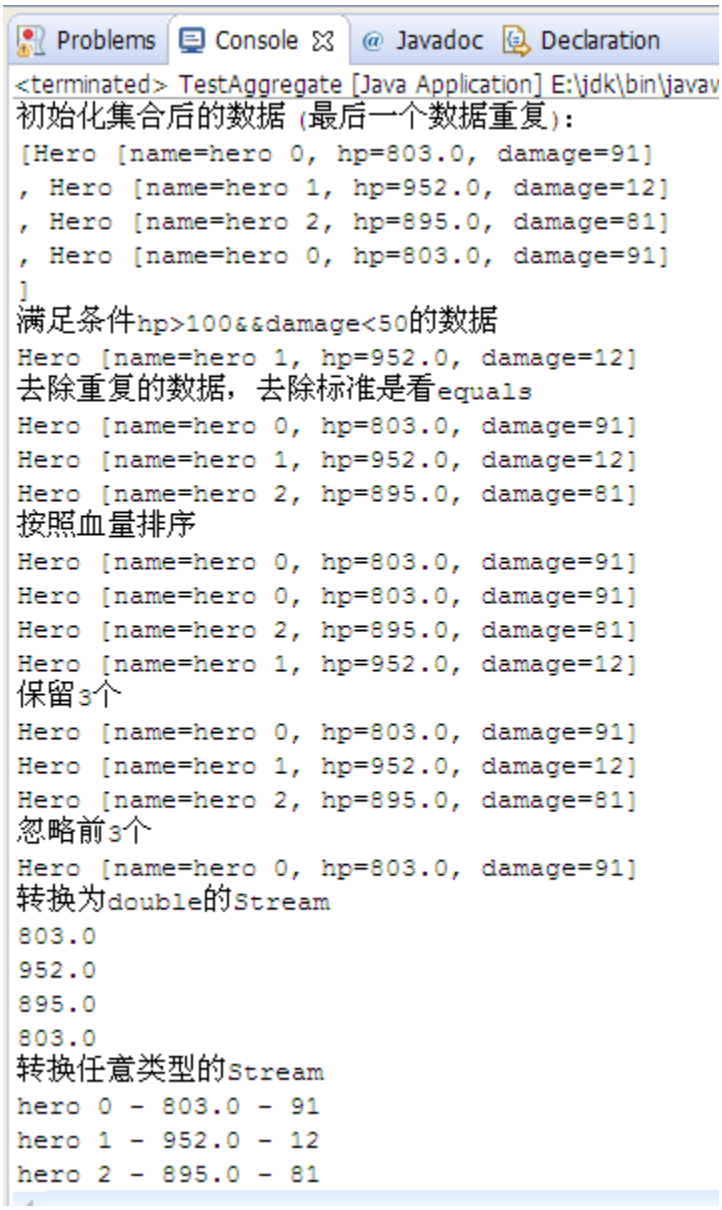
```
Stream.of(hs)
```

```
package lambda;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestAggregate {

    public static void main(String[] args) {
```

```
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        //管道源是集合

        heros
                .stream()
                .forEach(h->System.out.println(h.name));


        //管道源是数组

        Hero hs[] = heros.toArray(new Hero[heros.size()]);
        Arrays.stream(hs)
                .forEach(h->System.out.println(h.name));

    }
}
```

中间操作

每个中间操作，又会返回一个 Stream，比如.filter()又返回一个 Stream，中间操作是“懒”操作，并不会真正进行遍历。
中间操作比较多，主要分两类
对元素进行筛选 和 转换为其他形式的流
对元素进行筛选：
filter 匹配
distinct 去除重复(根据 equals 判断)
sorted 自然排序
sorted(Comparator<T>) 指定排序
limit 保留
skip 忽略
转换为其他形式的流
mapToDouble 转换为 double 的流
map 转换为任意类型的流




```java
package charactor;

public class Hero implements Comparable<Hero>{
    public String name;
    public float hp;

    public int damage;

    public Hero(){

    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```
        this.name = name;
    }
    public float getHp() {
        return hp;
    }
    public void setHp(float hp) {
        this.hp = hp;
    }
    public int getDamage() {
        return damage;
    }
    public void setDamage(int damage) {
        this.damage = damage;
    }
    public Hero(String name) {
        this.name =name;
    }
    //初始化 name,hp,damage 的构造方法
    public Hero(String name,float hp, int damage) {
        this.name =name;
        this.hp = hp;
        this.damage = damage;
    }

    @Override
    public int compareTo(Hero anotherHero) {
        if(damage<anotherHero.damage)
            return 1;
        else
            return -1;
    }

    @Override
    public String toString() {
        return "Hero [name=" + name + ", hp=" + hp + ",
damage=" + damage + "]\r\n";
    }

}
package lambda;

import java.util.ArrayList;
import java.util.List;
```

```
import java.util.Random;

import charactor.Hero;

public class TestAggregate {

    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }
        //制造一个重复数据
        heros.add(heros.get(0));
        System.out.println("初始化集合后的数据（最后一个数据重
复)：");
        System.out.println(heros);
        System.out.println("满足条件 hp>100&&damage<50 的数据
");

        heros
                .stream()
                .filter(h->h.hp>100&&h.damage<50)
                .forEach(h->System.out.print(h));


        System.out.println("去除重复的数据，去除标准是看
equals");
        heros
                .stream()
                .distinct()
                .forEach(h->System.out.print(h));
        System.out.println("按照血量排序");
        heros
                .stream()
                .sorted((h1,h2)->h1.hp>=h2.hp?1:-1)
                .forEach(h->System.out.print(h));
```

```
        System.out.println("保留 3 个");
        heros
                .stream()
                .limit(3)
                .forEach(h->System.out.print(h));

        System.out.println("忽略前 3 个");
        heros
                .stream()
                .skip(3)
                .forEach(h->System.out.print(h));

        System.out.println("转换为 double 的 Stream");
        heros
                .stream()
                .mapToDouble(Hero::getHp)
                .forEach(h->System.out.println(h));

        System.out.println("转换任意类型的 Stream");
        heros
                .stream()
                .map((h)-> h.name + " - " + h.hp + " - " +
h.damage)
                .forEach(h->System.out.println(h));

    }
}
```

结束操作

当进行结束操作后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。 结束操作不会返回 Stream，但是会返回 int、float、String、 Collection 或者像 forEach，什么都不返回,。
结束操作才真正进行遍历行为，前面的中间操作也在这个时候，才真正的执行。
常见结束操作如下：
forEach() 遍历每个元素
toArray() 转换为数组
min(Comparator<T>) 取最小的元素
max(Comparator<T>) 取最大的元素

count() 总数
findFirst() 第一个元素

```
<terminated> TestAggregate [Java Application] D:\
遍历集合中的每个数据
Hero [name=hero 0, hp=554.0, damage=15]
Hero [name=hero 1, hp=419.0, damage=86]
Hero [name=hero 2, hp=202.0, damage=44]
Hero [name=hero 3, hp=829.0, damage=32]
Hero [name=hero 4, hp=208.0, damage=27]
返回一个数组
[Hero [name=hero 0, hp=554.0, damage=15]
, Hero [name=hero 1, hp=419.0, damage=86]
, Hero [name=hero 2, hp=202.0, damage=44]
, Hero [name=hero 3, hp=829.0, damage=32]
, Hero [name=hero 4, hp=208.0, damage=27]
]
返回伤害最低的那个英雄
Hero [name=hero 0, hp=554.0, damage=15]
返回伤害最高的那个英雄
Hero [name=hero 1, hp=419.0, damage=86]
流中数据的总数
5
第一个英雄
Hero [name=hero 0, hp=554.0, damage=15]
```

```java
package lambda;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

import org.omg.Messaging.SYNC_WITH_TRANSPORT;

import charactor.Hero;

public class TestAggregate {

    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 5; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("遍历集合中的每个数据");

        heros
                .stream()
                .forEach(h->System.out.print(h));
```

```
        System.out.println("返回一个数组");
        Object[] hs= heros
                .stream()
                .toArray();
        System.out.println(Arrays.toString(hs));
        System.out.println("返回伤害最低的那个英雄");
        Hero minDamageHero =
                heros
                        .stream()
                        .min((h1,h2)->h1.damage-h2.damage)
                        .get();
        System.out.print(minDamageHero);
        System.out.println("返回伤害最高的那个英雄");

        Hero mxnDamageHero =
                heros
                        .stream()
                        .max((h1,h2)->h1.damage-h2.damage)
                        .get();
        System.out.print(mxnDamageHero);

        System.out.println("流中数据的总数");
        long count = heros
                .stream()
                .count();
        System.out.println(count);

        System.out.println("第一个英雄");
        Hero firstHero =
                heros
                        .stream()
                        .findFirst()
                        .get();

        System.out.println(firstHero);

    }
}
```

练习-聚合操作

首选准备 10 个 Hero 对象，hp 和 damage 都是随机数。
分别用传统方式和聚合操作的方式，把 hp 第三高的英雄名称打印出来

```
<terminated> TestAggregate [Java Application] E:\jdk\bin\javaw.e
初始化集合后的数据（最后一个数据重复）：
[Hero [name=hero 0, hp=114.0, damage=92]
, Hero [name=hero 1, hp=433.0, damage=55]
, Hero [name=hero 2, hp=691.0, damage=76]
, Hero [name=hero 3, hp=830.0, damage=48]
, Hero [name=hero 4, hp=979.0, damage=34]
, Hero [name=hero 5, hp=157.0, damage=27]
, Hero [name=hero 6, hp=177.0, damage=1]
, Hero [name=hero 7, hp=208.0, damage=2]
, Hero [name=hero 8, hp=29.0, damage=86]
, Hero [name=hero 9, hp=884.0, damage=43]
]
通过传统方式找出来的hp第三高的英雄名称是:hero 3
通过聚合操作找出来的hp第三高的英雄名称是:hero 3
```

```
package lambda;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;

import charactor.Hero;

public class TestAggregate {

    public static void main(String[] args) {
        Random r = new Random();
        List<Hero> heros = new ArrayList<Hero>();
        for (int i = 0; i < 10; i++) {
            heros.add(new Hero("hero " + i, r.nextInt(1000),
r.nextInt(100)));
        }

        System.out.println("初始化集合后的数据（最后一个数据重

复）：");

        System.out.println(heros);
```

```
        //传统方式
        Collections.sort(heros,new Comparator<Hero>() {
            @Override
            public int compare(Hero o1, Hero o2) {
                return (int) (o2.hp-o1.hp);
            }
        });

        Hero hero = heros.get(2);
        System.out.println("通过传统方式找出来的 hp 第三高的英雄
名称是:" + hero.name);


        //聚合方式
        String name =heros
                .stream()
                .sorted((h1,h2)->h1.hp>h2.hp?-1:1)
                .skip(2)
                .map(h->h.getName())
                .findFirst()
                .get();

        System.out.println("通过聚合操作找出来的 hp 第三高的英雄
名称是:" + name);

    }
}
```

网络编程

获取本机 IP 地址

```
InetAddress host = InetAddress.getLocalHost();
                String ip =host.getHostAddress();
                System.out.println("本机 ip 地址: " + ip);
```

使用 java 执行 ping 命令

```
package socket;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TestSocket {

    public static void main(String[] args) throws
IOException {

        Process p = Runtime.getRuntime().exec("ping " +
"192.168.2.106");
        BufferedReader br = new BufferedReader(new
InputStreamReader(p.getInputStream()));
        String line = null;
        StringBuilder sb = new StringBuilder();
        while ((line = br.readLine()) != null) {
            if (line.length() != 0)
                sb.append(line + "\r\n");
        }

        System.out.println("本次指令返回的消息是：");

        System.out.println(sb.toString());
    }

}
```

JAVA SOCKET 收发消息入门例子

建立连接

1. 服务端开启 8888 端口，并监听着，时刻等待着客户端的连接请求
2. 客户端知道服务端的 ip 地址和监听端口号，发出请求到服务端

客户端的端口地址是系统分配的，通常都会大于 1024

一旦建立了连接，服务端会得到一个新的 Socket 对象，该对象负责与客户端进行通信。

注意： 在开发调试的过程中，如果修改过了服务器 Server 代码，要关闭启动的 Server,否则新的 Server 不能启动，因为 8888 端口被占用了


```java
package socket;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public static void main(String[] args)  {
        try {
            //服务端打开端口 8888
            ServerSocket ss = new ServerSocket(8888);

            //在 8888 端口上监听，看是否有连接请求过来
            System.out.println("监听在端口号:8888");
            Socket s =  ss.accept();

            System.out.println("有连接过来" + s);

            s.close();
```

```
            ss.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

```
package socket;

import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {

    public static void main(String[] args)  {

        try {
            //连接到本机的8888端口
            Socket s = new Socket("127.0.0.1",8888);
            System.out.println(s);
            s.close();
        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

收发数字

一旦建立了连接，服务端和客户端就可以通过 Socket 进行通信了
1. 客户端打开输出流，并发送数字 110
2. 服务端打开输入流，接受数字 110，并打印

```
package socket;

import java.io.IOException;
import java.io.InputStream;
```

```java
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public static void main(String[] args) {
        try {

            ServerSocket ss = new ServerSocket(8888);


            System.out.println("监听在端口号:8888");

            Socket s = ss.accept();


            //打开输入流

            InputStream is = s.getInputStream();


            //读取客户端发送的数据

            int msg = is.read();

            //打印出来

            System.out.println(msg);
            is.close();


            s.close();
            ss.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
package socket;

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {
```

```
    public static void main(String[] args) {

        try {
            Socket s = new Socket("127.0.0.1", 8888);

            // 打开输出流
            OutputStream os = s.getOutputStream();

            // 发送数字110到服务端
            os.write(110);
            os.close();

            s.close();
        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

收发字符串

直接使用字节流收发字符串比较麻烦，使用数据流对字节流进行封装，这样收发字符串就容易了

1. 把输出流封装在 DataOutputStream 中
使用 writeUTF 发送字符串 "Legendary!"
2. 把输入流封装在 DataInputStream
使用 readUTF 读取字符串,并打印

```
package socket;

import java.io.DataInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public static void main(String[] args) {
```

```
        try {

            ServerSocket ss = new ServerSocket(8888);


            System.out.println("监听在端口号:8888");
            Socket s = ss.accept();

            InputStream is = s.getInputStream();


            //把输入流封装在DataInputStream
            DataInputStream dis = new DataInputStream(is);
            //使用readUTF 读取字符串
            String msg = dis.readUTF();
            System.out.println(msg);
            dis.close();
            s.close();
            ss.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
package socket;

import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class Client {

    public static void main(String[] args) {

        try {
            Socket s = new Socket("127.0.0.1", 8888);

            OutputStream os = s.getOutputStream();
```

```
            //把输出流封装在DataOutputStream 中
            DataOutputStream dos = new DataOutputStream(os);
            //使用writeUTF 发送字符串
            dos.writeUTF("Legendary!");
            dos.close();
            s.close();
        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

使用 Scanner

在上个步骤中，每次要发不同的数据都需要修改代码
可以使用 Scanner 读取控制台的输入，并发送到服务端，这样每次都可以发送不同的数据了。

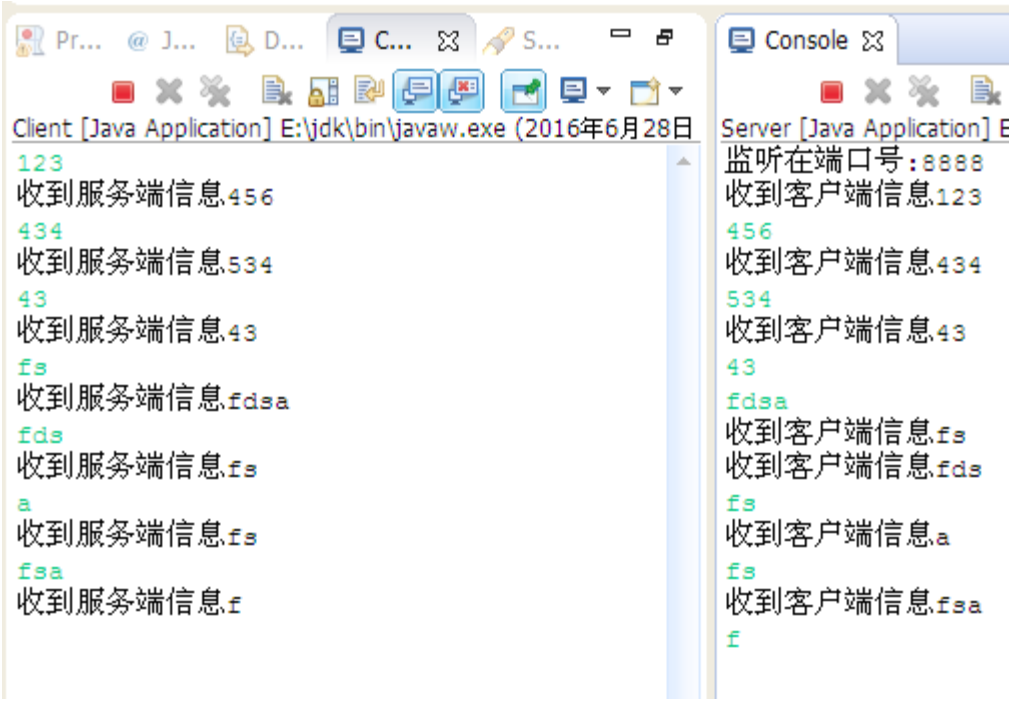
```
package socket;

import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class Client {

    public static void main(String[] args) {

        try {
            Socket s = new Socket("127.0.0.1", 8888);

            OutputStream os = s.getOutputStream();
            DataOutputStream dos = new DataOutputStream(os);
```

```
            //使用Scanner读取控制台的输入，并发送到服务端
            Scanner sc = new Scanner(System.in);

            String str = sc.next();
            dos.writeUTF(str);

            dos.close();
            s.close();
        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

练习-服务端和客户端互聊

前面部分的学习效果是服务端接受数据，客户端发送数据。

做相应的改动，使得服务端也能发送数据，客户端也能接受数据，并且可以一直持续下去



答案-服务端和客户端互聊

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class Server {

    public static void main(String[] args) {
        try {

            ServerSocket ss = new ServerSocket(8888);


            System.out.println("监听在端口号:8888");

            Socket s = ss.accept();

            InputStream is = s.getInputStream();
            DataInputStream dis = new DataInputStream(is);
            OutputStream os = s.getOutputStream();
            DataOutputStream dos = new DataOutputStream(os);

            while (true) {
                String msg = dis.readUTF();

                System.out.println("收到客户端信息"+msg);

                Scanner sc = new Scanner(System.in);
                String str = sc.next();
                dos.writeUTF(str);
            }

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

```java
package socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class Client {

    public static void main(String[] args) {

        try {
            Socket s = new Socket("127.0.0.1", 8888);

            OutputStream os = s.getOutputStream();
            DataOutputStream dos = new DataOutputStream(os);
            InputStream is = s.getInputStream();
            DataInputStream dis = new DataInputStream(is);

            while(true){
                Scanner sc = new Scanner(System.in);
                String str = sc.next();
                dos.writeUTF(str);
                String msg = dis.readUTF();

                System.out.println("收到服务端信息"+msg);

            }

        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

练习-聊天机器人数据库

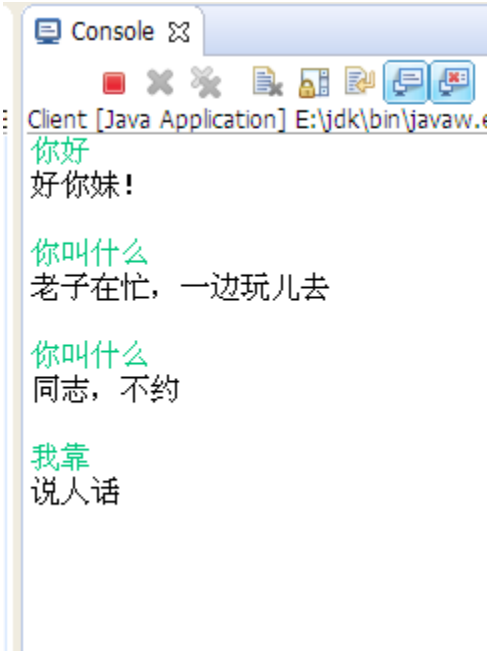
首先创建一个数据库 android(android 现在是手机操作系统，其本意是机器人的意思)

然后创建一个表 dictionary，字段：
id int
receive varchar(100)
response varchar(100)

receive 表示受到的信息
response 表示回应的信息

在这个表里准备一些数据：
你好 -> 好你妹！
你叫什么 -> 你想泡我啊？
打你哦 -> 来啊，来打我啊，不打有点挫
等等
等等

开发一个程序，当从 scanner 读取到消息，发给 Server 服务端，服务端用这个消息到表 dictionary 中找到对应的相应，返回出去。 看上去就像在自动回应一样。
如果一个 recieve 有多条 response，则随机返回一条



答案-聊天机器人数据库

```
create database android;
```

```
use android;

create table dictionary(
      id int AUTO_INCREMENT,
      receive varchar(100),
      response varchar(100),
      PRIMARY KEY (id)
)   DEFAULT CHARSET=utf8;

insert into dictionary values(null,'你好','好你妹！');
insert into dictionary values(null,'你叫什么','你想泡我啊？');
insert into dictionary values(null,'你叫什么','同志，不约');
insert into dictionary values(null,'打你哦','来啊，来打我啊，不打有点
挫');
```

```
package socket;

public class Dictionary {
    int id;
    String receive ;
    String response;
}
```

```
package socket;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import charactor.Hero;

public class DictionaryDAO {

    public DictionaryDAO() {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
```

```
    }

    public Connection getConnection() throws SQLException {
        return
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/and
roid?characterEncoding=UTF-8", "root",
                "admin");
    }

    public List<Dictionary> query(String recieve) {
        List<Dictionary> ds = new ArrayList<Dictionary>();

        String sql = "select * from dictionary where receive
= ? ";

        try (Connection c = getConnection();
PreparedStatement ps = c.prepareStatement(sql);) {
            ps.setString(1, recieve);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                Dictionary d = new Dictionary();
                int id = rs.getInt(1);
                String receive = rs.getString("receive");
                String response = rs.getString("response");
                d.id=id;
                d.receive=receive;
                d.response=response;
                ds.add(d);
            }
        } catch (SQLException e) {

            e.printStackTrace();
        }
        return ds;
    }

}


package socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
```

```
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Server {

    private static List<String> cannotUnderstand= new
ArrayList<>();
    static{
        cannotUnderstand.add("听求不懂啊");

        cannotUnderstand.add("说人话");

        cannotUnderstand.add("再说一遍？");

        cannotUnderstand.add("大声点");

        cannotUnderstand.add("老子在忙，一边玩儿去");

    }
    public static void main(String[] args) {
        try {

            ServerSocket ss = new ServerSocket(8888);


            System.out.println("监听在端口号:8888");
            Socket s = ss.accept();

            InputStream is = s.getInputStream();
            DataInputStream dis = new DataInputStream(is);
            OutputStream os = s.getOutputStream();
            DataOutputStream dos = new DataOutputStream(os);

            while (true) {
                String msg = dis.readUTF();
                System.out.println(msg);

                List<Dictionary> ds= new
DictionaryDAO().query(msg);
```

```
                String response = null;
                if(ds.isEmpty()){
                    Collections.shuffle(cannotUnderstand);
                    response = cannotUnderstand.get(0);
                }
                else{
                    Collections.shuffle(ds);
                    response = ds.get(0).response;
                }
                dos.writeUTF(response);
            }

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}


package socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class Client {

    public static void main(String[] args) {

        try {
            Socket s = new Socket("127.0.0.1", 8888);

            OutputStream os = s.getOutputStream();
            DataOutputStream dos = new DataOutputStream(os);
            InputStream is = s.getInputStream();
            DataInputStream dis = new DataInputStream(is);
```

```
            while(true){
                Scanner sc = new Scanner(System.in);
                String str = sc.nextLine();
                dos.writeUTF(str);
                String msg = dis.readUTF();
                System.out.println(msg);
                System.out.println();

            }

        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

开发多线程聊天程序

同时收发消息

在练习-服务端和客户端互聊 中，只能一人说一句，说了之后，必须等待另一个人的回复，才能说下一句。

这是因为接受和发送都在主线程中，不能同时进行。 为了实现同时收发消息，基本设计思路是把收发分别放在不同的线程中进行

1. SendThread 发送消息线程
2. RecieveThread 接受消息线程
3. Server 一旦接受到连接，就启动收发两个线程
4. Client 一旦建立了连接，就启动收发两个线程

```
package socket;

import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;
```

```java
public class SendThread extends Thread{

    private Socket s;

    public SendThread(Socket s){
        this.s = s;
    }
    public void run(){
        try {
            OutputStream os = s.getOutputStream();
            DataOutputStream dos = new DataOutputStream(os);

            while(true){
                Scanner sc = new Scanner(System.in);
                String str = sc.next();
                dos.writeUTF(str);
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

```java
package socket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

public class RecieveThread extends Thread {

    private Socket s;

    public RecieveThread(Socket s) {
        this.s = s;
    }
```

```
    public void run() {
        try {
            InputStream is = s.getInputStream();

            DataInputStream dis = new DataInputStream(is);
            while (true) {
                String msg = dis.readUTF();
                System.out.println(msg);
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}


package socket;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public static void main(String[] args) {
        try {

            ServerSocket ss = new ServerSocket(8888);


            System.out.println("监听在端口号:8888");

            Socket s = ss.accept();


            //启动发送消息线程

            new SendThread(s).start();

            //启动接受消息线程

            new RecieveThread(s).start();
```

```
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

```
package socket;

import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {

    public static void main(String[] args) {

        try {
            Socket s = new Socket("127.0.0.1", 8888);


            // 启动发送消息线程

            new SendThread(s).start();

            // 启动接受消息线程

            new RecieveThread(s).start();

        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

反射

什么是类对象

在理解类对象之前，先说我们熟悉的对象之间的区别：
garen 和 teemo 都是 **Hero 对象**，他们的区别在于，各自有不**同的名称，血量，伤害值**。

然后说说类之间的区别
Hero 和 Item 都是类，他们的区别在于有**不同的方法，不同的属性**。

类对象，就是用于描述这种类，都有什么属性，什么方法的

获取类对象

获取类对象有 3 种方式
1. Class.forName
2. Hero.class
3. new Hero().getClass()

在一个 JVM 中，一种类，只会有一个类对象存在。所以以上三种方式取出来的类对象，都是一样的。

注：准确的讲是一个 ClassLoader 下，一种类，只会有一个类对象存在。通常一个 JVM 下，只会有一个 ClassLoader。因为还没有引入 ClassLoader 概念， 所以暂时不展开了。

```
package reflection;

import charactor.Hero;

public class TestReflection {

        public static void main(String[] args) {
                        String className = "charactor.Hero";
                        try {
                                Class
pClass1=Class.forName(className);
                                Class pClass2=Hero.class;
                                Class pClass3=new Hero().getClass();
                                System.out.println(pClass1==pClass2);
                                System.out.println(pClass1==pClass3);
```

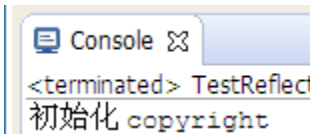
```
                        } catch (ClassNotFoundException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
}
```

获取类对象的时候，会导致类属性被初始化

为 Hero 增加一个静态属性,并且在静态初始化块里进行初始化，参考 类属性初始化。

```
static String copyright;
static {
        System.out.println("初始化  copyright");
        copyright = "版权由 Riot Games 公司所有";
}
```

无论什么途径获取类对象，都会导致静态属性被初始化，而且只会执行一次。（除了直接使用 Class c = Hero.class 这种方式，这种方式不会导致静态属性被初始化）




```
package charactor;

public class Hero {
        public String name;
        public float hp;
        public int damage;
        public int id;

        static String copyright;

        static {
                System.out.println("初始化 copyright");
                copyright = "版权由 Riot Games 公司所有";
        }

}
```

```
package reflection;

import charactor.Hero;

public class TestReflection {

        public static void main(String[] args) {
                        String className = "charactor.Hero";
                        try {
                                Class
pClass1=Class.forName(className);
                                Class pClass2=Hero.class;
                                Class pClass3=new Hero().getClass();
                        } catch (ClassNotFoundException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
        }
}
```

练习-在静态方法上加 synchronized，同步对象是什么？

在之前有一个练习，练习-在类前面加修饰符 synchronized

在对象方法前，加上修饰符 synchronized，同步对象是当前实例。
那么如果在类方法前，加上修饰符 synchronized，同步对象是什么呢？

编写代码进行验证

当 **synchronized** 修饰静态方法的时候， 同步对象就是这个类的类对象。

如代码中的例子，当第一个线程进入 **method1** 的时候，需要占用 **TestReflection.class** 才能执行。

第二个线程进入 **method2** 的时候进去不，只有等第一个线程释放了对 **TestReflection.class** 的占用，才能够执行。 反推过来，第二个线程也是需要占用 **TestReflection.class**。 那么 **TestReflection.class** 就是 **method2** 的同步对象。

换句话说，静态方法被修饰为 **synchronized** 的时候，其同步对象就是当前类的类对象。

```
package reflection;

public class TestReflection {
    public static void main(String[] args) throws
InterruptedException {
        Thread t1= new Thread(){
            public void run(){
                //调用method1
                TestReflection.method1();
            }
        };
        t1.setName("第一个线程");
        t1.start();

        //保证第一个线程先调用method1
        Thread.sleep(1000);

        Thread t2= new Thread(){
            public void run(){
                //调用method2
```

```
                TestReflection.method2();
            }
        };
        t2.setName("第二个线程");
        t2.start();
    }

    public static void method1() {
        synchronized (TestReflection.class) {
            // 对于method1 而言，同步对象是
TestReflection.class，只有占用 TestReflection.class 才可以执行
到这里

System.out.println(Thread.currentThread().getName() + " 进入
了 method1 方法");
            try {
                System.out.println("运行 5 秒");
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static synchronized void method2() {
        // 对于mehotd2 而言，必然有个同步对象，通过观察发现，当某
个线程在method1 中，占用了 TestReflection.class 之后
        // 就无法进入 method2，推断出，method2 的同步对象，就是
TestReflection.class
        System.out.println(Thread.currentThread().getName() +
" 进入了 method2 方法");
```

```
        try {
            System.out.println("运行 5 秒");
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

创建一个对象

```
package reflection;
import java.lang.reflect.Constructor;
import charactor.Hero;
public class TestReflection {

        public static void main(String[] args) {
                //传统的使用 new 的方式创建对象
                Hero h1 =new Hero();
                h1.name = "teemo";
                System.out.println(h1);

                try {
                        //使用反射的方式创建对象
                        String className = "charactor.Hero";
                        //类对象
                        Class pClass=Class.forName(className);
                        //构造器
                        Constructor c= pClass.getConstructor();
                        //通过构造器实例化
                        Hero h2= (Hero) c.newInstance();
                        h2.name="gareen";
                        System.out.println(h2);
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

练习-通过配置文件获取对象

首先准备一个文本文件：hero.config。 在这个文件中保存类的全名称，可以是 charactor.APHero 或者是 charactor.ADHero

接着设计一个方法叫做：

```
public static Hero getHero()
```

在这个方法中，读取 hero.config 的数据，取出其中的类名，根据类名实例化出对象，然后返回对象。

```
package charactor;

public class APHero extends Hero {

        public void magicAttack() {
                System.out.println("进行魔法攻击");
        }

}
```

```
package charactor;

public class ADHero extends Hero {

        public void physicAttack() {
                System.out.println("进行物理攻击");
        }

}
```

答案-通过配置文件获取对象

```
Hero h = getHero();
System.out.println(h);
```

通过打印 h，可以发现，当配置文件里的内容发生变化的时候，就会得到不同的对象。

源代码不需要发生任何变化，只需要修改配置文件，就可以导致程序的逻辑发生变化， 这是一种基于配置的编程思想。

Spring 框架中的 IOC 和 DI 的底层就是基于这样的机制实现的。

```
package reflection;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

import charactor.Hero;

public class TestReflection {
    public static void main(String[] args) throws
InterruptedException {
        Hero h = getHero();
        System.out.println(h);
    }

    public static Hero getHero() {

        File f = new File("E:/project/j2se/hero.config");

        try (FileReader fr = new FileReader(f)) {
            String className = null;
            char[] all = new char[(int) f.length()];
            fr.read(all);
            className = new String(all);
            Class clazz=Class.forName(className);
            Constructor c= clazz.getConstructor();
            Hero h= (Hero) c.newInstance();
            return h;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }

    }
}
```

访问属性

Hero.java

为了访问属性，把 name 修改为 public。
对于 private 修饰的成员，需要使用 setAccessible(true)才能访问和修改。不在此知识点讨论。

```
package charactor;

public class Hero {
        public String name;
        public float hp;
        public int damage;
        public int id;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public Hero(){

        }
        public Hero(String string) {
                name =string;
        }

        @Override
        public String toString() {
                return "Hero [name=" + name + "]";
        }
        public boolean isDead() {
                // TODO Auto-generated method stub
                return false;
        }
        public void attackHero(Hero h2) {
                System.out.println(this.name+ " 正在攻击 " +
h2.getName());
        }

}
```

TestRelection

通过反射修改属性的值

```
package reflection;

import java.lang.reflect.Field;

import charactor.Hero;

public class TestReflection {

    public static void main(String[] args) {
        Hero h =new Hero();

        //使用传统方式修改name的值为garen

        h.name = "garen";
        try {

            //获取类Hero的名字叫做name的字段

            Field f1= h.getClass().getDeclaredField("name");

            //修改这个字段的值

            f1.set(h, "teemo");

            //打印被修改后的值

            System.out.println(h.name);

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

getField 和 getDeclaredField 的区别

getField 和 getDeclaredField 的区别
这两个方法都是用于获取字段
getField 只能获取 public 的，包括从父类继承来的字段。
getDeclaredField 可以获取本类所有的字段，包括 private 的，但是不能获取继承来的字段。（注： 这里只能获取到 private 的字段，但并不能访问该 private 字段的值,除非加上 setAccessible(true)）

反射机制 调用方法

首先为 Hero 的 name 属性，增加 setter 和 getter
通过反射机制调用 Hero 的 setName

```
package charactor;

public class Hero {
    public String name;
    public float hp;
    public int damage;
    public int id;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Hero(){

    }
    public Hero(String string) {
        name =string;
    }

    @Override
    public String toString() {
        return "Hero [name=" + name + "]";
    }
    public boolean isDead() {
        // TODO Auto-generated method stub
        return false;
    }
    public void attackHero(Hero h2) {
        // TODO Auto-generated method stub

    }

}
package reflection;

import java.lang.reflect.Method;

import charactor.Hero;
```

```
public class TestReflection {

    public static void main(String[] args) {
        Hero h = new Hero();

        try {
            // 获取这个名字叫做 setName，参数类型是 String 的方法
            Method m = h.getClass().getMethod("setName",
String.class);
            // 对 h 对象，调用这个方法

            m.invoke(h, "盖伦");

            // 使用传统的方式，调用 getName 方法
            System.out.println(h.getName());

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

练习-调用方法

继续上一个练习 练习-通过配置文件获取对象，把 hero.config 改动成为支持如下格式：

```
charactor.APHero
garen
charactor.ADHero
teemo
```

首先根据这个配置文件，使用反射实例化出两个英雄出来。
然后通过反射给这两个英雄设置名称，接着再通过反射，调用第一个英雄的 attackHero 方法，攻击第二个英雄

```
package charactor;
```

```java
public class Hero {
    public String name;
    public float hp;
    public int damage;
    public int id;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Hero(){

    }
    public Hero(String string) {
        name =string;
    }

    @Override
    public String toString() {
        return "Hero [name=" + name + "]";
    }
    public boolean isDead() {
        // TODO Auto-generated method stub
        return false;
    }
    public void attackHero(Hero h2) {
        System.out.println(this.name+ " 正在攻击 " +
h2.getName());
    }

}

package charactor;

public class APHero extends Hero {

    public void magicAttack() {
        System.out.println("进行魔法攻击");
    }
```

```
}
```

```
package charactor;

public class ADHero extends Hero {

    public void physicAttack() {
        System.out.println("进行物理攻击");
    }

}
```

答案-调用方法

```
package reflection;

import java.io.File;
import java.io.FileReader;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

import charactor.Hero;

public class TestReflection {
    public static void main(String[] args) throws
InterruptedException {
        File f = new File("E:/project/j2se/hero.config");

        try (FileReader fr = new FileReader(f)) {
            String fileContent = null;
            char[] all = new char[(int) f.length()];
            fr.read(all);
            fileContent = new String(all);
            String[] cs = fileContent.split("\r\n");
            String hero1className = cs[0];
            String hero1Name = cs[1];
            String hero2className = cs[2];
            String hero2Name = cs[3];
```

```
            //根据反射，获取hero1，并且给hero1的name字段赋值
            Class hero1Class = Class.forName(hero1className);
            Constructor hero1Constructor =
hero1Class.getConstructor();
            Object hero1 = hero1Constructor.newInstance();
            Field hero1NameField =
hero1Class.getField("name");
            hero1NameField.set(hero1, hero1Name);

            //根据反射，获取hero2,并且给hero2的name字段赋值
            Class hero2Class = Class.forName(hero2className);
            Constructor hero2Constructor =
hero2Class.getConstructor();
            Object hero2 = hero2Constructor.newInstance();
            Field hero2NameField =
hero2Class.getField("name");
            hero2NameField.set(hero2, hero2Name);

            //根据反射，获取attackHero方法，并且调用hero1的这
个方法，参数是hero2
            Method attackHeroMethod =
hero1Class.getMethod("attackHero", Hero.class);
            attackHeroMethod.invoke(hero1, hero2);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

反射机制 有什么用

业务类

首先准备两个业务类，这两个业务类很简单，就是各自都有一个业务方法，分别打印不同的字符串

```
package reflection;
```

```
public class Service1 {

	public void doService1(){
		System.out.println("业务方法 1");
	}
}
```

```
package reflection;

public class Service2 {

	public void doService2(){
		System.out.println("业务方法 2");
	}
}
```

非反射方式

当需要从第一个业务方法切换到第二个业务方法的时候，使用非反射方式，必须修改代码，并且重新编译运行，才可以达到效果

```
package reflection;

public class Test {

	public static void main(String[] args) {
		new Service1().doService1();
	}
}
```

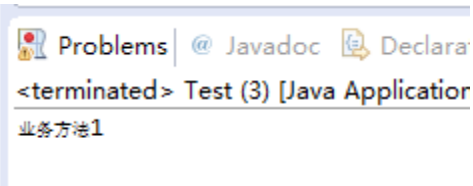
```
package reflection;

public class Test {

	public static void main(String[] args) {
//		new Service1().doService1();
		new Service2().doService2();
	}
}
```

反射方式

使用反射方式，首先准备一个配置文件，就叫做 spring.txt 吧，放在 src 目录下。
里面存放的是类的名称，和要调用的方法名。
在测试类 Test 中，首先取出类名称和方法名，然后通过反射去调用这个方法。

当需要从调用第一个业务方法，切换到调用第二个业务方法的时候，不需要修改一行代码，也不需要重新编译，只需要修改配置文件 spring.txt，再运行即可。

这也是 Spring 框架的最基本的原理，只是它做的更丰富，安全，健壮。




```
class=reflection.Service1
method=doService1
```

```
package reflection;

import java.io.File;
import java.io.FileInputStream;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.util.Properties;

public class Test {

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void main(String[] args) throws Exception
{


        //从 spring.txt 中获取类名称和方法名称
        File springConfigFile = new
File("e:\\project\\j2se\\src\\spring.txt");
        Properties springConfig= new Properties();
        springConfig.load(new
FileInputStream(springConfigFile));
        String className = (String)
springConfig.get("class");
```

```
        String methodName = (String)
springConfig.get("method");

        //根据类名称获取类对象
        Class clazz = Class.forName(className);
        //根据方法名称，获取方法对象
        Method m = clazz.getMethod(methodName);
        //获取构造器
        Constructor c = clazz.getConstructor();
        //根据构造器，实例化出对象
        Object service = c.newInstance();
        //调用对象的指定方法
        m.invoke(service);

    }
}
```

注解

基本内置注解

@Override

@Override 用在方法上，表示这个方法重写了父类的方法，如 toString()。
如果父类没有这个方法，那么就无法编译通过，如例所示，在 fromString()方法上

加上@Override 注解，就会失败，因为 Hero 类的父类 Object，并没有 fromString 方法
代码比较复制代码

```
1 package annotation;
2
3 public class Hero {
4
5         String name;
6         @Override
7         public String toString(){
8                 return name;
9         }
10        @Override
11        public String fromString(){
12                return name;
13        }
14}
```

@Deprecated

@Deprecated 表示这个方法已经过期，不建议开发者使用。(暗示在将来某个不确定的版本，就有可能会取消掉)

如例所示，开地图这个方法 hackMap，被注解为过期，在调用的时候，就会受到提示

```
package annotation;

public class Hero {

        String name;

        @Deprecated
        public void hackMap(){

        }
        public static void main(String[] args) {
                new Hero().hackMap();
        }

}
```

@SuppressWarnings

@SuppressWarnings Suppress 英文的意思是抑制的意思，这个注解的用处是忽略警告信息。
比如大家使用集合的时候，有时候为了偷懒，会不写泛型，像这样：

```
List heros = new ArrayList();
```

那么就会导致编译器出现警告，而加上

```
@SuppressWarnings({ "rawtypes", "unused" })
```

就对这些警告进行了抑制，即忽略掉这些警告信息。
@SuppressWarnings 有常见的值，分别对应如下意思
1.deprecation：使用了不赞成使用的类或方法时的警告(使用@Deprecated 使得编译器产生的警告)；
2.unchecked：执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型; 关闭编译器警告
3.fallthrough：当 Switch 程序块直接通往下一种情况而没有 Break 时的警告;
4.path：在类路径、源文件路径等中有不存在的路径时的警告;
5.serial：当在可序列化的类上缺少 serialVersionUID 定义时的警告;
6.finally：任何 finally 子句不能正常完成时的警告;
7.rawtypes 泛型类型未指明
8.unused 引用定义了，但是没有被使用
9.all：关于以上所有情况的警告。
代码比较复制代码

```
1 package annotation;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Hero {
7         String name;
8         @SuppressWarnings({ "rawtypes", "unused" })
9         public static void main(String[] args) {
10                 List heros = new ArrayList();
11         }
12
13}
```

@SafeVarargs

@SafeVarargs 这是 1.7 之后新加入的基本注解. 如例所示，当使用可变数量的参数的时候，而参数的类型又是泛型 T 的话，就会出现警告。 这个时候，就使用 @SafeVarargs 来去掉这个警告

@SafeVarargs 注解只能用在参数长度可变的方法或构造方法上，且方法必须声明为 static 或 final，否则会出现编译错误。一个方法使用@SafeVarargs 注解的前提是，开发人员必须确保这个方法的实现中对泛型类型参数的处理不会引发类型安全问题。

以上解释很复杂，**我也没搞明白，请忽略**，往下学习。。。
代码比较复制代码

```
1 package annotation;
2
3 public class Hero {
4         String name;
5
6         @SafeVarargs
7         public static <T> T getFirstOne(T... elements) {
8                 return elements.length > 0 ? elements[0] : null;
9         }
10}
```

@FunctionalInterface

@FunctionalInterface 这是 Java1.8 新增的注解，用于约定函数式接口。
函数式接口概念： 如果接口中只有一个抽象方法（可以包含多个默认方法或多个 static 方法），该接口称为函数式接口。函数式接口其存在的意义，主要是配合 Lambda 表达式 来使用。

如例所示， AD 接口只有一个 adAttack 方法，那么就可以被注解为 @FunctionalInterface,而 AP 接口有两个方法 apAttack()和 apAttack2(),那么就不能被注解为函数式接口

```
package annotation;

@FunctionalInterface
public interface AD {
        public void adAttack();
}

package annotation;
```

```
@FunctionalInterface
public interface AP {
        public void apAttack();
        public void apAttack2();
}
```

自定义注解

在本例中，把数据库连接的工具类 **DBUtil** 改造成为注解的方式，来举例演示怎么自定义注解以及如何解析这些自定义注解

非注解方式 DBUtil

通常来讲，在一个基于 JDBC 开发的项目里，都会有一个 DBUtil 这么一个类，在这个类里统一提供连接数据库的 IP 地址，端口，数据库名称， 账号，密码，编码方式等信息。如例所示，在这个 DBUtil 类里，这些信息，就是以属性的方式定义在类里的。

大家可以运行试试，运行结果是获取一个连接数据库 test 的连接 Connection 实例。
注：运行需要用到连接 mysql 的 jar 包，如果没有，可在右侧下载

```
package util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBUtil {
    static String ip = "127.0.0.1";
    static int port = 3306;
    static String database = "test";
    static String encoding = "UTF-8";
    static String loginName = "root";
    static String password = "admin";
    static{
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
```

```
        }
    }

    public static Connection getConnection() throws
SQLException {
        String url =
String.format("jdbc:mysql://%s:%d/%s?characterEncoding=%s",
ip, port, database, encoding);
        return DriverManager.getConnection(url, loginName,
password);
    }
    public static void main(String[] args) throws
SQLException {
        System.out.println(getConnection());
    }
}
```

自定义注解@JDBCConfig

接下来，就要把 DBUtil 这个类改造成为支持自定义注解的方式。 首先创建一个注解 JDBCConfig

1. 创建注解类型的时候即不使用 class 也不使用 interface,而是使用@interface

```
public @interface JDBCConfig
```

2. 元注解
@Target({METHOD,TYPE}) 表示这个注解可以用用在类/接口上，还可以用在方法上
@Retention(RetentionPolicy.RUNTIME) 表示这是一个运行时注解，即运行起来之后，才获取注解中的相关信息，而不像基本注解如@Override 那种不用运行，在编译时 eclipse 就可以进行相关工作的编译时注解。
@Inherited 表示这个注解可以被子类继承
@Documented 表示当执行 javadoc 的时候，本注解会生成相关文档
请在学习完本知识点最后一个步骤解析注解之后，再查看 元注解，做更详尽的学习。

3. 注解元素，这些注解元素就用于存放注解信息，在解析的时候获取出来

```
String ip();
int port() default 3306;
String database();
String encoding();
```

```
String loginName();
String password();
```

```
package anno;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;

import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({METHOD,TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface JDBCConfig {
    String ip();
    int port() default 3306;
    String database();
    String encoding();
    String loginName();
    String password();
}
```

注解方式 DBUtil

有了自定义注解@JDBCConfig 之后，我们就把非注解方式 DBUtil 改造成为注解方式 DBUtil。
如例所示，数据库相关配置信息本来是以属性的方式存放的，现在改为了以注解的方式，提供这些信息了。

注：目前只是以注解的方式提供这些信息，但是还没有解析，接下来进行解析

```
package util;

import anno.JDBCConfig;

@JDBCConfig(ip = "127.0.0.1", database = "test", encoding = "UTF-8",
loginName = "root", password = "admin")
public class DBUtil {
        static {
                try {
```

```
                Class.forName("com.mysql.jdbc.Driver");
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }

}
```

解析注解

接下来就通过反射，获取这个 DBUtil 这个类上的注解对象

```
JDBCConfig config = DBUtil.class.getAnnotation(JDBCConfig.class);
```

拿到注解对象之后，通过其方法，获取各个注解元素的值：

```
String ip = config.ip();
int port = config.port();
String database = config.database();
String encoding = config.encoding();
String loginName = config.loginName();
String password = config.password();
```

后续就一样了，根据这些配置信息得到一个数据库连接 Connection 实例。
注： 运行需要用到连接 mysql 的 jar 包，如果没有，可在右侧下载

```
package util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import anno.JDBCConfig;

@JDBCConfig(ip = "127.0.0.1", database = "test", encoding =
"UTF-8", loginName = "root", password = "admin")
public class DBUtil {
    static {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
```

```
        }
    }

    public static Connection getConnection() throws
SQLException, NoSuchMethodException, SecurityException {
        JDBCConfig config =
DBUtil.class.getAnnotation(JDBCConfig.class);

        String ip = config.ip();
        int port = config.port();
        String database = config.database();
        String encoding = config.encoding();
        String loginName = config.loginName();
        String password = config.password();

        String url =
String.format("jdbc:mysql://%s:%d/%s?characterEncoding=%s",
ip, port, database, encoding);
        return DriverManager.getConnection(url, loginName,
password);
    }

    public static void main(String[] args) throws
NoSuchMethodException, SecurityException, SQLException {
        Connection c = getConnection();
        System.out.println(c);
    }
}
```

元注解概念

在讲解**元注解**概念之前，我们先建立**元数据**的概念。 元数据在英语中对应单词 **metadata**, metadata 在 wiki 中的解释是：

Metadata is data [information] that provides information about other data

为其他数据提供信息的数据

这样元注解就好理解了，元注解 meta annotation 用于注解 自定义注解 的注解。

元注解有这么几种：

@Target
@Retention
@Inherited
@Documented
@Repeatable (java1.8 新增)

接下来挨个讲解

@Target

@Target 表示这个注解能放在什么位置上，是只能放在类上？还是即可以放在方法上，又可以放在属性上。自定义注解@JDBCConfig 这个注解上的@Target 是：@Target({METHOD,TYPE})，表示他可以用在方法和类型上（类和接口），但是不能放在属性等其他位置。 可以选择的位置列表如下：
ElementType.TYPE：能修饰类、接口或枚举类型
ElementType.FIELD：能修饰成员变量
ElementType.METHOD：能修饰方法
ElementType.PARAMETER：能修饰参数
ElementType.CONSTRUCTOR：能修饰构造器
ElementType.LOCAL_VARIABLE：能修饰局部变量
ElementType.ANNOTATION_TYPE：能修饰注解
ElementType.PACKAGE：能修饰包

```
package anno;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;

import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({METHOD,TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface JDBCConfig {
    String ip();
    int port() default 3306;
    String database();
    String encoding();
    String loginName();
    String password();
}
```

@Retention

@Retention 表示生命周期，自定义注解@JDBCConfig 上的值是 RetentionPolicy.RUNTIME，表示可以在运行的时候依然可以使用。@Retention 可选的值有 3 个：

RetentionPolicy.SOURCE：注解只在源代码中存在，编译成 class 之后，就没了。@Override 就是这种注解。
RetentionPolicy.CLASS：注解在 java 文件编程成.class 文件后，依然存在，但是运行起来后就没了。@Retention 的默认值，即当没有显式指定@Retention 的时候，就会是这种类型。
RetentionPolicy.RUNTIME：注解在运行起来之后依然存在，程序可以通过反射获取这些信息，自定义注解@JDBCConfig 就是这样。
大家可以试试把自定义注解@JDBCConfig 的@Retention 改成其他两种，并且运行起来，看看有什么不同

```
package anno;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;

import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({METHOD,TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented

public @interface JDBCConfig {
    String ip();
    int port() default 3306;
    String database();
    String encoding();
    String loginName();
    String password();
}
```

@Inherited

@Inherited 表示该注解具有继承性。如例，设计一个 DBUtil 的子类，其 getConnection2 方法，可以获取到父类 DBUtil 上的注解信息。

```
package util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
import anno.JDBCConfig;

public class DBUtilChild extends DBUtil {

    public static Connection getConnection2() throws
SQLException, NoSuchMethodException, SecurityException {
        JDBCConfig config =
DBUtilChild.class.getAnnotation(JDBCConfig.class);
        String ip = config.ip();
        int port = config.port();
        String database = config.database();
        String encoding = config.encoding();
        String loginName = config.loginName();
        String password = config.password();

        String url =
String.format("jdbc:mysql://%s:%d/%s?characterEncoding=%s",
ip, port, database, encoding);
        return DriverManager.getConnection(url, loginName,
password);
    }

    public static void main(String[] args) throws
NoSuchMethodException, SecurityException, SQLException {
        Connection c = getConnection2();
        System.out.println(c);
    }
}
```

@Documented

@Documented 如图所示， 在用 javadoc 命令生成 API 文档后，DBUtil 的文档里会出现该注解说明。

注： 使用 eclipse 把项目中的.java 文件导成 API 文档步骤：

1. 选中项目
2. 点开菜单 File
3. 点击 Export
4. 点开 java->javadoc->点 next
5. 点 finish

@Repeatable (java1.8 新增)

当没有@Repeatable 修饰的时候，注解在同一个位置，只能出现一次，如例所示：
@JDBCConfig(ip = "127.0.0.1", database = "test", encoding = "UTF-8", loginName = "root", password = "admin")
@JDBCConfig(ip = "127.0.0.1", database = "test", encoding = "UTF-8", loginName = "root", password = "admin")
重复做两次就会报错了。
使用@Repeatable 之后，再配合一些其他动作，就可以在同一个地方使用多次了。
如何使用@Repeatable 单独拿出来讲解：@Repeatable 运用举例

```
package util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import anno.JDBCConfig;

@JDBCConfig(ip = "127.0.0.1", database = "test", encoding =
"UTF-8", loginName = "root", password = "admin")
```

```
@JDBCConfig(ip = "127.0.0.1", database = "test", encoding =
"UTF-8", loginName = "root", password = "admin")
public class DBUtil {
    static {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static Connection getConnection() throws
SQLException, NoSuchMethodException, SecurityException {
        JDBCConfig config =
DBUtil.class.getAnnotation(JDBCConfig.class);
        System.out.println(config);

        String ip = config.ip();
        int port = config.port();
        String database = config.database();
        String encoding = config.encoding();
        String loginName = config.loginName();
        String password = config.password();

        String url =
String.format("jdbc:mysql://%s:%d/%s?characterEncoding=%s",
ip, port, database, encoding);
        return DriverManager.getConnection(url, loginName,
password);
    }

    public static void main(String[] args) throws
NoSuchMethodException, SecurityException, SQLException {
        Connection c = getConnection();
        System.out.println(c);
    }
}
```

@Repeatable 运用举例

比如在练习练习-查找文件内容 中有一个要求，即查找文件后缀名是.java 的文件，我们把部分代码修改为注解，并且使用@Repeatable 这个元注解来表示，文件后缀名的范围可以是 java, html, css, js 等等。

为了紧凑起见，把注解作为内部类的形式放在一个文件里。
1. 注解 FileTypes，其 value()返回一个 FileType 数组
2. 注解 FileType，其@Repeatable 的值采用 FileTypes
3. 运用注解：在 work 方法上重复使用多次@FileType 注解
4. 解析注解： 在 work 方法内，通过反射获取到本方法上的 FileType 类型的注解数组，然后遍历本数组

```
package annotation;
import static java.lang.annotation.ElementType.METHOD;

import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public class FindFiles {
    @Target( METHOD)
    @Retention( RetentionPolicy.RUNTIME )
    public @interface FileTypes {
        FileType[] value();
    }

    @Target(  METHOD )
    @Retention( RetentionPolicy.RUNTIME )
    @Repeatable( FileTypes.class )
    public @interface FileType {
        String value();
    };

    @FileType( ".java" )
    @FileType( ".html" )
    @FileType( ".css" )
    @FileType( ".js" )
    public void work(){

        try {
            FileType[] fileTypes=
this.getClass().getMethod("work").getAnnotationsByType(FileT
ype.class);

            System.out.println("将从如下后缀名的文件中查找文件内

容");

            for (FileType fileType : fileTypes) {
                System.out.println(fileType.value());
```

```
            }
            System.out.println("查找过程略。。。");
        } catch (NoSuchMethodException | SecurityException e)
{
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new FindFiles().work();
    }
}
```

仿 HIBERNATE 注解

hibernate 两种配置方式

hibernate 有两种配置方式，分别是*.hbm.xml 配置方式 和注解方式。 虽然方式不一样，但是都是用于解决如下问题：

1. 当前类是否实体类
2. 对应的表名称
3. 主键对应哪个属性， 自增长策略是什么，对应字段名称是什么
4. 非主键属性对应字段名称是什么

接下来，我会做一套仿 hibernate 的注解，并且在一个实体类 Hero 上运用这些注解，并通过反射解析这些注解信息，来解决上述的问题

自定义 hibernate 注解

参考 hibernate 的 注解配置方式， 自定义 5 个注解，分别对应 hibernate 中用到的注解：

hibernate_annotation.MyEntity 对应 javax.persistence.Entity
hibernate_annotation.MyTable 对应 javax.persistence.Table
hibernate_annotation.MyId 对应 javax.persistence.Id
hibernate_annotation.MyGeneratedValue 对应 javax.persistence.GeneratedValue
hibernate_annotation.MyColumn 对应 javax.persistence.Column

```
package hibernate_annotation;

import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyEntity {

}
```

```
package hibernate_annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTable {

        String name();
}
```

```
package hibernate_annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyId {

}
```

```
package hibernate_annotation;
```

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyGeneratedValue {
        String strategy();
}
```

```
package hibernate_annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyColumn {
        String value();
}
```

运用在 Hero 对象上

像以注解方式配置 Product 类 那样，在 Hero 类上运用这些自定义注解：
当注解的方法是 value 的时候，给这个注解赋值时，本来应该是：

```
@MyColumn(value="name_")
```

现在可以简略一点，写为

```
@MyColumn("name_")
```

只有当名称是 value 的时候可以这样，其他名称如 name,stratgy 等不行

```java
package pojo;

import hibernate_annotation.MyColumn;
import hibernate_annotation.MyEntity;
import hibernate_annotation.MyGeneratedValue;
import hibernate_annotation.MyId;
import hibernate_annotation.MyTable;

@MyEntity
@MyTable(name="hero_")
public class Hero {
    private int id;
    private String name;
    private int damage;
    private int armor;

    @MyId
    @MyGeneratedValue(strategy = "identity")
    @MyColumn("id_")
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    @MyColumn("name_")
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @MyColumn("damage_")
    public int getDamage() {
        return damage;
    }
    public void setDamage(int damage) {
        this.damage = damage;
    }
    @MyColumn("armor_")
    public int getArmor() {
        return armor;
    }
    public void setArmor(int armor) {
```

```
        this.armor = armor;
    }

}
```

解析注解

创建一个解析类 ParseHibernateAnnotation ，获取 Hero 类上配置的注解信息，其运行结果如图所示。
思路如下：
1. 首先获取 Hero.class 类对象
2. 判断本类是否进行了 MyEntity 注解
3. 获取注解 MyTable
4. 遍历所有的方法，如果某个方法有 MyId 注解，那么就记录为主键方法 primaryKeyMethod
5. 把主键方法的自增长策略注解 MyGeneratedValue 和对应的字段注解 MyColumn 取出来，并打印
6. 遍历所有非主键方法，并且有 MyColumn 注解的方法，打印属性名称和字段名称的对应关系。

```
Hero类是实体类
其对应的表名是:hero_
找到主键: id
其自增长策略是: identity
对应数据库中的字段是: id_
其他非主键属性分别对应的数据库字段如下:
属性: name        对应的数据库字段是:name_
属性: damage      对应的数据库字段是:damage_
属性: armor       对应的数据库字段是:armor_
```

```
import java.lang.reflect.Method;

import hibernate_annotation.MyColumn;
import hibernate_annotation.MyEntity;
import hibernate_annotation.MyGeneratedValue;
import hibernate_annotation.MyId;
import hibernate_annotation.MyTable;
import pojo.Hero;

public class ParseHibernateAnnotation {

    public static void main(String[] args) {

        Class<Hero> clazz = Hero.class;
        MyEntity myEntity = (MyEntity) clazz.getAnnotation(MyEntity.class);
        if (null == myEntity) {
```

```
            System.out.println("Hero 类不是实体类");
        } else {
            System.out.println("Hero 类是实体类");
            MyTable myTable= (MyTable) clazz.getAnnotation(MyTable.class);
            String tableName = myTable.name();
            System.out.println("其对应的表名是:" + tableName);
            Method[] methods =clazz.getMethods();
            Method primaryKeyMethod = null;
            for (Method m: methods) {
                MyId myId = m.getAnnotation(MyId.class);
                if(null!=myId){
                    primaryKeyMethod = m;
                    break;
                }
            }

            if(null!=primaryKeyMethod){
                System.out.println("找到主键:"+method2attribute( primaryKeyMethod.getName() ));
                MyGeneratedValue myGeneratedValue =
                        primaryKeyMethod.getAnnotation(MyGeneratedValue.class);
                System.out.println("其自增长策略是:"+myGeneratedValue.strategy());
                MyColumn myColumn = primaryKeyMethod.getAnnotation(MyColumn.class);
                System.out.println("对应数据库中的字段是:"+myColumn.value());
            }
            System.out.println("其他非主键属性分别对应的数据库字段如下:");
            for (Method m: methods) {
                if(m==primaryKeyMethod){
                    continue;
                }
                MyColumn myColumn = m.getAnnotation(MyColumn.class);
                //那些 setter 方法上是没有 MyColumn 注解的
                if(null==myColumn)
                    continue;
                System.out.format("属性:\t%s\t对应的数据库字段是:%s%n",method2attribute(m.getName()),myColumn.value());

            }

        }

    }
```

```
    private static String method2attribute(String methodName) {
        String result = methodName; ;
        result = result.replaceFirst("get", "");
        result = result.replaceFirst("is", "");
        if(result.length()<=1){
            return result.toLowerCase();
        }
        else{
            return result.substring(0,1).toLowerCase() + result.substring(1,result.length());
        }

    }
}
```

END

2017/12/12