



# 从C到JAVA

## 阶梯教程

© 2017-2020, 宿宝臣



subaochen@126.com

<http://dz.sdut.edu.cn/blog/subaochen>

<https://github.com/subaochen/java-tutorial-examples>

# 前言

本书作为为数不多的涉及 **Java8+** 的教材，期望能够帮助读者在更高的平台上（本书放弃了 **Java5** 之前的部分特性）尽快掌握 **Java** 语言及其编程技能。本书的目标是在尽量短的时间内，比如 48 个学时，帮助 **Java** 初学者掌握基本的 **Java** 语法和编程思路。

一般的，理工科院校讲授的第一门计算机程序设计语言是 **C** 语言，因此大家在学习（自学或者教学）**Java** 语言时已经具有了一定的 **C** 语言基础。而目前常见的 **Java** 语言教材或者辅导材料基本上都是零基础开始讲授 **Java** 语言程序设计的，完全无视大家的 **C** 语言基础，导致教与学活动中的很大浪费，也不利于突出 **Java** 教学活动的重点。因此，本书不是一本零基础的 **Java** 教程，本书在很多章节将 **C** 语言和 **Java** 做了对比，引导读者从 **C** 语言转换到 **Java** 语言上来，作者期望读者在 **C** 语言的基础上能够实现 **Java** 语言的快速入门。

## 本书的特点

本书注重培养良好的思维习惯、编程风格，而不是简单的知识传授。很多教材的示例代码过于随意，可能是作者觉得示例代码过于短小，不值得精心雕琢吧。区别于大多数教材的是，本书在编写示例代码时，全部经过作者的一手调试，并尽力保证类名、变量名等命名习惯以及编程风格（包括注释风格）接近工业标准，以帮助读者”先入为主“的了解正确的 **Java** 编程姿态。如果我们学习 **Java** 的目的是对接工业化应用，作者认为这样可以事半功倍。

本书注重从实战中学习 **Java** 的知识点，本书尽力做到每一个概念、每一个知识点都是可以验证的，书中的每一个例子都是完整可运行的，都经过作者的亲自调试。本书的所有示例均可以从<https://github.com/subaochen/java-tutorial-examples> 下载，作者也会根据读者的反馈不断完善和调整示例程序。

本书通过思维导图的方式绘出了 **Java** 的各个知识点，并针对每个知识点设计了练习题，帮助读者更好的了解这些知识点及其在实际中的应用方式。

本书的例子均遵循 **google** 的编码规范（参考附录），希望读者在阅读本书示例代码的同时能够直观的感受 **google** 编码规范并逐步养成遵守编码规范的好习惯。

## 如何阅读本书

本书不强调从一开始就掌握 Java 的每个技术细节，而是强调“先跑起来！”，即首先动手写出一个可以运行的 Java 应用程序，并理解这个应用程序为什么能够跑起来，然后再逐步扩充到细节问题。也就是说，在实践中体味和掌握技术细节，在实践中逐步形成良好的编程风格和思维习惯。

作者建议的阅读方式是：尽快、尽早的下载本书配套的示例代码，在阅读到例题的时候，直接用 **Idea** 打开相应的项目即可查看和运行。同时，建议不要拘泥于作者提供的示例代码，读者完全可以也应该不断尝试修改并测试。编程能力就是在不断尝试中发展起来的。

本书也特别强调循序渐进的学习路线和知识的前后衔接以及连贯性。虽然 Java 语言经常存在一个概念贯穿前后的情况，但是本书强调先掌握这个概念的部分特征，随着学习的深入，将在后续章节逐步展开，并给出恰当的交叉索引将知识点逐步串联起来。

本书的章节标题如果是 \* 开头的为较高要求内容，读者可以有选择的阅读，或者作为进阶的阅读材料。

## 你适合阅读本书吗

本书不是一本零基础的 Java 语言教程。本书假设读者已经有了一些 C 语言的基础，掌握了 C 语言的基本语法。另外，本书在很多方面比较深入的探讨了 Java 编程的理念和最佳实践，也可以作为工程技术人员学习 Java 语言或者进一步理解 Java 语言的材料。

## 本书的体例

### 印刷约定

字体	意义	示例
<i>AbCd123</i> 斜体	文件名、路径名、域名等	<code>ls -l filename</code>
<b>AbCd123</b> 加粗	在终端输入的命令等	<code>subaochen_desktop% su</code>
等宽字体	示例代码、代码片段等	<code>public class MyClass...</code>

### 图形标识

本书使用了如下的图形标识帮助读者更好的区分和了解知识点所在：



需要注意的知识点。



工程实践中常用的技巧。



容易出错的地方，需要特别小心。

## 联系作者

您可以通过我的博客获得最新的消息：<http://dz.sdut.edu.cn/blog/subaochen>，或者给我发 Email: [subaochen@126.com](mailto:subaochen@126.com)。本书中的示例代码可以从<http://github.com/subaochen/java-tutorial-examples>下载，也欢迎读者不吝指教，在 github 提交 PR，或者直接发邮件讨论亦可。您可以在本书的不同章节看到如何获得源代码的相关提示。

## 本书是如何写成的

本书全部使用开源（Open Source）软件完成：

- **Linux**，本书所有稿件和源代码均在 **Linux** 下完成。
- **git**，本书的写作过程全程通过 **git** 进行版本控制，也借助于 **git** 在办公室、家和旅程中实现文档的同步，收益良多。
- **Lyx**(<http://www.lyx.org>)，优秀的 **Latex** 前端可视化工具，最新版本（本书写作时是 2.2）配合 **xetex**、**CTex** 可以很好的支持中文处理。
- **graphviz**，灵活而强大的代码绘图工具，本书部分流程图是使用 **graphviz** 绘制的。
- **Shutter**(<http://shutter-project.org>)，**Linux** 下面优秀的截图工具。
- **ArgoUML**(<http://argouml.tigris.org>)，优秀的开源 **UML** 工具。
- **umbrello**(<https://umbrello.kde.org>)，优秀的开源 **UML** 工具，本书部分 **UML** 图是使用 **umbrello** 绘制的。
- **Dia**，优秀的开源绘制工具，堪称 **Linux** 下面写作绘图的“瑞士军刀”，本书的大部分流程图、框图和 **UML** 图都是用 **Dia** 绘制的。

- Inkscape, 优秀的开源矢量图绘制工具, 本书的大部分矢量图是用 Inkscape 绘制的。
- vym(<http://www.insilmaril.de/vym/>), 思维导图绘制工具, 本书所有思维导图都是使用 vym 绘制的。

## 致谢

感谢山东理工大学电气与电子学院对本书的资助, 感谢各位同仁给予的帮助和指点, 感谢爱妻程玉华承担了大部分家务, 让我有充足的时间专心写作; 感谢实验室的小伙伴们, 尤其是胡安禄同学和徐千惠同学, 帮助画出了部分示例图; 感谢李松阳同学在繁忙的工作之余认真校对, 从标点符号到遣词造句都给出了很多具体的意见和建议。

特别的, 要感谢我的女儿宿佳敏。写作本书时她正在读高中, 作者答应在这个寒假结束时写完这本书送给她。我做到了, 信守一个父亲的诺言。作者坚信, 这是最长情的陪伴和对她学习的最好帮助。

在写作本书过程中, 作者查阅了大量 Java 教材和网络资料, 所引用或依据的精彩论述和案例尽量在文中标出, 以便读者参照和比较, 同时对原作者表示深深的敬意和感谢!

也感谢所有为开源软件作出贡献的人们! 二十年前, 当年懵懂的作者决定彻底拥抱开源软件时, 为了安装一个 Linux 系统曾经不眠不休两天两夜; 回头望, 开源软件蓬勃发展二十年, 值得欣慰, 值得弹冠相庆! 没有 Linux, 没有 Latex, 没有 Lyx, 没有 Dia, 这本书也不可能如此顺利的完稿。致敬, Open Source!

宿宝臣

2017 年 2 月

于山东理工大学 1 号实验楼

# 目录

前言	i
第一章 参数化类型：泛型	4
1.1 泛型的引入	4
1.1.1 从一个小例子说起:Box	4
1.1.2 泛型化的 GenericBox	6
1.1.3 Java 泛型的一般语法	7
1.1.4 泛型类型参数的常见形式	9
1.2 泛型的常见形态	9
1.2.1 泛型类	9
1.2.2 泛型接口	9
1.2.3 泛型方法	11
1.2.4 泛型数组	12
1.3 泛型的高级用法	13
1.3.1 通配符	13
1.3.2 泛型的上下界	13
1.3.3 泛型方法使用泛型类的类型参数	17
1.4 使用泛型的注意事项	18
1.5 类型擦除	19
1.5.1 擦除类定义中的类型参数	19
1.5.2 擦除方法定义中的类型参数	20
1.5.3 桥接方法和泛型的多态	21
1.6 泛型的原生态类型	22
第二章 集合类	23
2.1 集合的概念	23
2.2 集合类框架的接口	24
2.2.1 Collection 接口	24
2.2.2 Set 接口	25

2.2.3 List 接口 . . . . .	30
2.2.4 Queue 接口 . . . . .	37
2.2.5 Map 接口 . . . . .	40
2.2.6 Iterator 接口 . . . . .	46
2.2.7 Collection 的 addAll 方法 . . . . .	50
2.3 集合类中的泛型: PECS 原则 . . . . .	51
2.3.1 extends . . . . .	52
2.3.2 super . . . . .	53
2.3.3 PECS . . . . .	54
2.4 集合类的一般实用原则和注意事项 . . . . .	56
<b>第三章 lambda 表达式</b>	<b>57</b>
3.1 为什么引入 lambda 表达式? . . . . .	57
3.2 lambda 表达式的类型 . . . . .	64
3.3 lambda 表达式的应用场景 . . . . .	64
3.4 集合的流式处理 . . . . .	65
3.5 lambda 表达式的作用范围 . . . . .	68
<b>第四章 多线程 Java 程序设计</b>	<b>70</b>
4.1 线程的基本概念 . . . . .	70
4.2 线程的创建 . . . . .	71
4.2.1 使用 sleep 暂时中断线程的执行 . . . . .	73
4.2.2 使用 interrupt 终止线程的执行 . . . . .	75
4.2.3 使用匿名内部类创建线程 . . . . .	79
4.3 线程间的数据共享和协作 . . . . .	79
4.3.1 使用 synchronized 保护竞争资源 . . . . .	80
4.3.2 使用 ThreadLocal 隔离竞争资源 . . . . .	86
4.3.3 使用 wait、notify 实现多线程的同步和协调 . . . . .	90
<b>第五章 网络编程</b>	<b>95</b>
5.1 网络通讯基础知识 . . . . .	95
5.1.1 TCP . . . . .	96
5.1.2 UDP . . . . .	96
5.1.3 IP 地址 . . . . .	96
5.1.4 端口 (port) . . . . .	96
5.2 使用 URL . . . . .	97
5.2.1 什么是 URL? . . . . .	97
5.2.2 创建 URL . . . . .	98

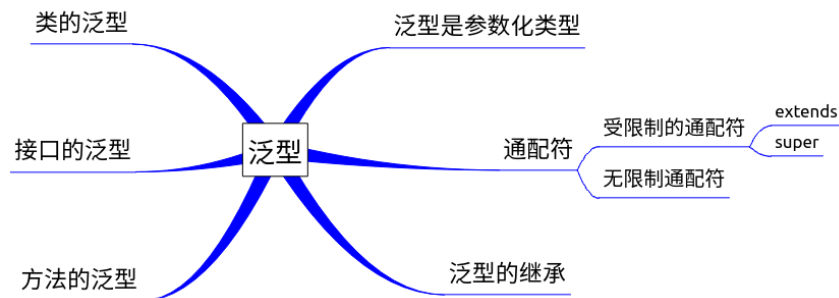
---

5.2.3 解析 URL . . . . .	100
5.2.4 读取 URL . . . . .	101
5.3 Socket 编程 . . . . .	102
5.3.1 什么是 Socket? . . . . .	103
5.3.2 使用 Socket . . . . .	103
5.3.3 * 编写健壮的服务器端 Socket 应用 . . . . .	108



# 第一章 参数化类型：泛型

通常的教材一般先讲述集合类，再讲述泛型，这是按照技术点出现的顺序来安排的：的确，集合类从 JDK 1.0 就有了，泛型是从 JDK 1.5 才出现的。但是，这样的教学安排会导致学生可能养成不良的编程习惯（使用原生态泛型），因此本书特地将泛型部分内容安排到集合类的前面讲述。



## 1.1 泛型的引入

### 1.1.1 从一个小例子说起:Box

假设有一个 Box 类（泛型是个框，神马都能装），我们希望这个 Box 能装任何的对象，因此定义其属性 value 的类型为 Object。其定义见代码清单1.1

代码清单 1.1: Box.java

```
1 package cn.edu.sdut.softlab;
2
3 public class Box {
4     private Object value;
5
6     public Object getValue() {
7         return this.value;
8     }
9
10    public void setValue(Object value) {
11        this.value = value;
12    }
13 }
```

```
12     }
13
14     public static void main(String[] args) {
15         Box box = new Box();
16         box.setValue(123); // ❶
17         System.out.println("456 + box' value = " + (456 + (Integer) box.getValue())); //
            ❶
18         box.setValue("123"); // ❷
19         System.out.println("456 + box' value = " + (456 + (Integer.valueOf(box.getValue()
            ().toString()))));
20         System.out.println("456 + box' value = " + (456 + (Integer) box.getValue())); //
            ❸
21     }
22 } //
```

❶ 这里利用了 Java 的自动装箱机制：整数 123 自动转换为 Integer 对象

❷ 我们知道 box 的 value 是一个 Integer，因此在这里进行了强制类型转换

❸ 字符串 123 作为 setValue 的参数是可以的

❹ 字符串对象无法自动转换为 Integer 对象，因此运行时抛出异常

当我们运行 Box 类时结果如下：

```
456 + box' value = 579
```

```
456 + box' value = 579
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
```

```
at cn.edu.sdut.softlab.Box.main(Box.java:20)
```

```
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
...
```

```
at java.lang.reflect.Method.invoke(Method.java:498)
```

```
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)
```

Box 类在编译时并没有提示任何错误<sup>1</sup>，但是运行时报告强制类型转换错误，这是不希望看到的，但是又无能为力：属性 value 是 Object 类型的，因此任何类型的对象都可以装进 box 对象中，因此我们无法预知 box 中到底装了什么类型的对象，也就无法在编译阶段检查 box 中的类型是否匹配应用场景了，这为程序的运行埋下了隐患<sup>2</sup>。

不过，我们也不是完全不能避免这种情况，比如我们可以设计一个 IntegerBox 专门装整数，设计一个 StringBox 专门装字符串，代码示例：

<sup>1</sup>随着学习的深入我们会发现，Java 编译器正在变得越来越聪明，即 Java 编译器尽量在编译阶段发现问题，这样就不至于在运行时才报告错误（这是异常处理机制的责任）。运行时发现错误，也许出错已经很久了，也许是很久以前的代码，只有通过搜索代码才能找到问题所在，耗时耗力。尽早发现错误，最好是编译阶段就发现并报告错误，泛型技术的引入也是为了达到这个目标。

<sup>2</sup>强制类型转换总是可能带来风险的，因此要格外谨慎对待，尤其是向下的强制类型转换，往往会带来程序的运行不稳定。

```
1 class IntegerBox {
2     private Integer value;
3     ... // setValue, getValue
4 }
5
6 class StringBox {
7     private String value;
8     ... // setValue, getValue
9 }
```

这样的结果是我们的代码变得冗长和重复，并且，要穷尽所有的情况或者不断跟踪需求的变化，真的是一件很吃力的事情。

### 1.1.2 泛型化的 GenericBox

泛型是解决上述问题的“魔术”。Java 从 1.5 开始引入了泛型的概念，我们可以这样定义 Box 类（为了区别起见，我们把这个泛型 Box 叫做 GenericBox，参见代码清单1.2）。

代码清单 1.2: GenericBox.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-23.
5  */
6 public class GenericBox<T> {
7     private T value;
8
9     public T getValue() {
10         return this.value;
11     }
12
13     public void setValue(T value) {
14         this.value = value;
15     }
16
17     public static void main(String[] args) {
18         GenericBox<Integer> box = new GenericBox<Integer>();
19         box.setValue(123); //❶
20         System.out.println("456 + box' value = " + (456 + box.getValue())); //❷
21         //box.setValue("123"); //❸
22         GenericBox<String> sbox = new GenericBox<String>(); //❹
23         sbox.setValue("123");
24         System.out.println("456 + box' value = " + (456 + (Integer.valueOf(box.getValue()
25             ()))));
26     }
27 }
```

- ❶ 这里利用了 Java 的自动装箱机制：整数 123 自动转换为 Integer 对象
- ❷ box 的 value 是一个整数，因此这里无须进行强制类型转换
- ❸ box 的 value 是整数类型的，因此在这里使用字符串 123 是非法的
- ❹ 创建一个使用字符串的 sbbox 对象

从形式上看，GenericBox 和 Box 的差别有两点：

所有在 Box 中出现 Object 的地方都代替以 T。你可能会迷惑，T 不是标准的 JDK 中的类，也不是我们自定义的类，Java 将如何识别 T 这个类呢？秘密在第二点。

在 GenericBox 类名后面使用了一对尖括号，其中列出了在代码中用到的 T，也就是说，在这对尖括号中的字母可以在代码中像类那样使用<sup>3</sup>。

图1.1清楚的表达了泛型类 GenericBox 和普通类 Box 的差异。

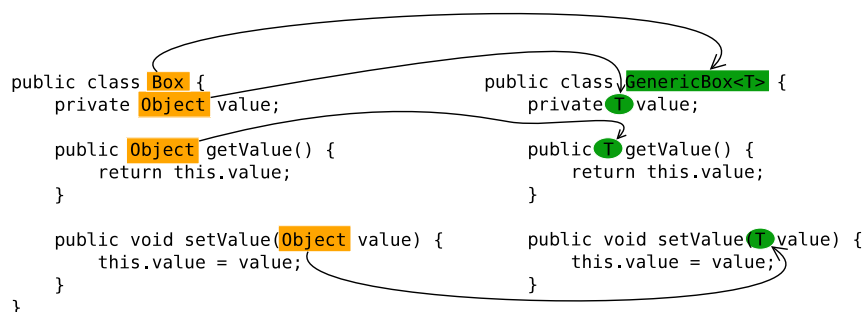


图 1.1: GenericBox 和 Box 的对比

### 1.1.3 Java 泛型的一般语法

每种泛型 (generic type) 定义一组参数化的类型 (parameterized type)，Java 泛型的一般语法形式如下：

```
class name<T1, T2, ... Tn>
```

其中，T1、T2 等称为类型参数 (type parameter)，这也是为什么泛型被称为参数化类型的原因：在实际引用泛型类的时候，可以通过传递实际的类的方式确定泛型类内部真正使用的类，如图1.2所示，非常像函数调用时形参和实参的匹配。

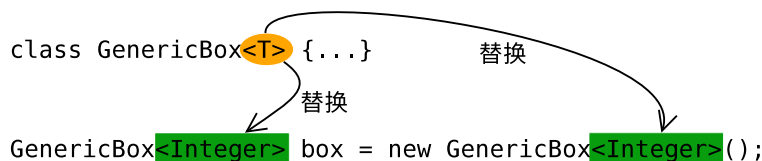


图 1.2: GenericBox 的用法示例

<sup>3</sup>粗略的可以这样认为，但是有例外，参见类型擦除机制 [?] 的相关说明。



要区分两个重要的概念：参数化的类型（parameterized type）和类型参数（type parameter）：参数化的类型就是泛型，我们之前接触到的类型（类、接口）是没有参数的<sup>a</sup>，因此一种类型（类、接口）只能处理一种类型的数据，现在有了泛型技术，类型（类、接口）可以通过在尖括号中定义类型参数（type parameter）“泛化”要处理的数据类型，这样类型（类、接口）的数据处理能力就大大增强了。

<sup>a</sup>每个泛型都定义了一个原生态类型（raw type），即不带任何实际类型参数的泛型名称。例如 `GenericBox<T>` 相对应的原生态类型是 `GenericBox`。原生态类型就像从类型声明中删除了所有泛型信息一样。本书不再展开讨论原生态类型，因为在编程实践中应该优先考虑使用泛型，尽量避免使用原生态类型。如果使用原生态类型，就失去了泛型在安全性和表述性方面的所有优势 [?, 第 23 条]。

如何确定尖括号中的类型参数需要多少个呢？这取决于类中需要多少种不同的数据类型，或者说，在泛型中，有多少种数据类型（类、接口）可以进行通用化处理，就需要多少个类型参数，这一点也很像函数设计的方法：在函数中需要多少个变量，就设置多少个函数参数。

**例 1.1.** 使用多个类型参数的泛型 `Box`，参见代码清单1.3。

代码清单 1.3: `GenericMultiBox.java`

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-24.
5  */
6 public class GenericMultiBox<K, V> {
7     private K key;
8     private V value;
9
10    public GenericMultiBox(K key, V value) {
11        this.key = key;
12        this.value = value;
13    }
14
15    public void info() {
16        System.out.println(key + ">=" + value);
17    }
18
19    public static void main(String[] args) {
20        GenericMultiBox<String,Integer> box = new GenericMultiBox<String, Integer>("age"
21            , 21);
22        box.info();
23    }
```

### 1.1.4 泛型类型参数的常见形式

泛型的类型参数不是一个已经预定义的类，你可以使用任意的字符串来表示类型参数，就像是函数参数的名字一样，只要是合法的变量名都可以作为泛型的类型参数。但是人们也形成了一些约定俗成的用法，遵守这些约定俗成的用法有助于理解别人写的代码，也有助于别人更好的理解你的代码：

- E - 元素 (Element 的第一个字母，在 Java 集合类框架中大量使用)
- K - 索引 (Key 的第一个字母)
- V - 值 (Value 的第一个字母)
- N - 数字 (Number 的第一个字母)
- T - 类 (Type 的第一个字母)
- X - 类
- S,U,V 等. - 第 2 个，第三个，第 4 个类等

## 1.2 泛型的常见形态

### 1.2.1 泛型类

其实我们在 section §1.1 中讲述的就是泛型类，也是最常见的泛型的形态，不再赘述。

**练习 1.1.** 使用泛型定义二叉树的节点 Node 类 [解答 ?? [在第 ??页]]。

### 1.2.2 泛型接口

接口是纯的抽象类，因此泛型接口和泛型类用法是一样的。

**例 1.2.** 接口上的泛型<sup>4</sup>。

**代码设计** 本例首先定义了一个很常见的 Generator 接口，用于规范如何获取下一个元素，参见代码清单1.4，然后给出了一个 generator 接口的具体实现 FruitGenerator，可以获取（产生）下一个水果的名称，参见代码清单1.5。

---

<sup>4</sup>本例借鉴了 <https://segmentfault.com/a/1190000002646193>，感谢原作者的辛勤付出！

代码清单 1.4: Generator.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-24.
5  */
6 public interface Generator<T> {
7     public T next();
8 }
```

代码清单 1.5: FruitGenerator.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.Random;
4
5 /**
6  * Created by subaochen on 16-12-24.
7  */
8 public class FruitGenerator implements Generator<String> {
9
10     private String[] fruits = new String[]{"Apple", "Banana", "Pear"};
11
12     @Override
13     public String next() {
14         Random rand = new Random();
15         return fruits[rand.nextInt(3)];
16     }
17
18     public static void main(String[] args) {
19         FruitGenerator generator = new FruitGenerator();
20         System.out.println(generator.next());
21         System.out.println(generator.next());
22         System.out.println(generator.next());
23         System.out.println(generator.next());
24     }
25 }
```

**运行结果** 在 Idea 中运行 FruitGenerator，可能的结果如下（每次运行结果可能有所不同）：

```
Pear
Banana
Pear
Pear
```

**练习 1.2.** 编写一个 `Generator` 接口的实现, 用于获取下一个连续的整数, 假设从 100 开始计算。

### 1.2.3 泛型方法

泛型作用在类上, 表示类中的属性或者使用到的变量可以泛型化 (即抽象化), 很自然的, 泛型作用在方法上, 即泛型方法是指方法中使用到的变量可以泛型化 (抽象化)。方法中可以抽象出来的变量只有形式参数和返回值, 如图1.3所示, 我们可以在一对尖括号中声明在形式参数和返回值中可能使用到的类型参数。我们可以把“类型参数”看做数学上方程的“自变量”, 在方法中我们把所有可变的因素抽象出来放到类型参数列表中, 就像是在数学上把所有可变的因素抽象出来看做自变量一样。泛型方法使得我们编写的方法可以适用于多种数据类型, 因此我们在编写方法时应该尽量使用泛型方法。可以这么说, 如果该方法可能不同的场合下使用两种以上不同的数据类型, 泛型方法就有存在的价值。

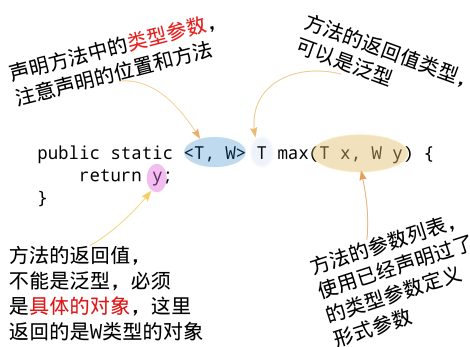


图 1.3: 泛型方法的常见形态

在这里要特别注意声明类型参数的位置和方法:

- 声明类型参数列表的方法和类中一样, 都是使用一对尖括号包围起来。有几个“自变量”, 我们就声明几个类型参数。尖括号是 `Java` 编译器识别类型参数的标识, 即只要在尖括号内部的字符串列表都会被当做类型参数对待。有时你可能会想, 去掉尖括号及其类型参数列表可以吗? 尝试一下会发现, 这会导致语法错误:

**can not resolve symbol 'T', 'W'**

这是很自然的: 去掉了尖括号的类型参数列表, 返回值的类型 `T` 和两个形式参数的类型 `T`、`W` 在哪里定义的呢? 因此, 尖括号及其类型参数列表是不能省略掉的, 并且声明的位置也不能搞错。

- 泛型方法的类型参数列表是在方法的返回值前面声明的<sup>5</sup>。
- 调用泛型方法时, 类型参数列表一般不需要给出, 比如可以直接调用泛型方法: `choise(123, "a string")`, `Java` 编译器可以根据参数的类型自动推断所对应的类型参数的类

<sup>5</sup> 不然呢? 你能想出更合理的位置来声明泛型方法的类型参数列表吗?



型。但是如果 Java 编译器无法自动推断，则需要明确给出泛型方法的类型参数列表的实际类型，比如 `GenericMethodTest.<String, Integer>choise(123, "a string")`。

当然，图1.3不是唯一的泛型方法的形态，比如：

```
1 public <S, U, V> U methodName(S x, V y) {  
2     ....  
3     U u = ...;  
4     return u;  
5 }
```

### 例 1.3. 泛型方法示例

代码清单 1.6: GenericMethodTest.java

```
1 package cn.edu.sdut.softlab;  
2  
3 /**  
4  * Created by subaochen on 16-12-25.  
5  */  
6 public class GenericMethodTest {  
7     public static <T, W> T choise(W x, T y) {  
8         return y;  
9     }  
10  
11     public static void main(String[] args) {  
12         System.out.println(choise(3,5));  
13         System.out.println(choise("1123",456));  
14         System.out.println(choise(123, "a string"));  
15         System.out.println(GenericMethodTest.<String, Integer>choise(123, "a string"));  
16     }  
17 }
```

## 1.2.4 泛型数组

泛型数组没有什么特别的，是指数组的数据类型是泛型，即数组的每个数据元素是一个泛型。比如代码清单1.7中，我们在 `info` 方法中使用了一个泛型数组作为参数，这样 `info` 方法就可以打印出任意类型的数组元素了。

代码清单 1.7: GenericArrayTest.java

```
1 package cn.edu.sdut.softlab;  
2  
3 /**  
4  * Created by subaochen on 17-1-7.  
5  */  
6 public class GenericArrayTest {
```

```
7 public static void main(String[] args) {
8     Integer[] intArray = {1,2,3};
9     info(intArray);
10    String[] sArray = {"java","language"};
11    info(sArray);
12 }
13
14 public static <T> void info(T[] array) {
15     for(T t : array) {
16         System.out.println(t);
17     }
18 }
19 }
```

## 1.3 泛型的高级用法

### 1.3.1 通配符

通配符? 的意义表示任意的类型，一般用在方法的参数中，比如：

```
1 public void test(Box<?> box) {...}
```

表示 box 对象可以容纳任何类型的对象。

注意到 Box<Object> 和 Box<?> 是有区别的，Box<Object> 表示只能容纳 Object 类型的对象，而 Box<?> 则可以容纳任何类型的对象，比如 Object, Integer, String 等等。

通配符? 用在类型参数中是没有意义的，比如下面的类定义：

```
1 public class Test<?> {...}
```

试想，如果这样的定义是合法的，那么我们在类里面如何引用这个泛型通配符代表的数据类型呢？所以，这样的写法是一个语法错误。

### 1.3.2 泛型的上下界

很多时候，我们并不希望泛型类或者泛型方法的类型参数是任意的类，比如代码清单1.2中的 GenericBox 什么都能装，如果我们只希望 GenericBox 处理各种数字，该怎么办呢？Java 的 extends 和 super 关键字可以用来限制类型参数的上界和下界，如图1.4所示。

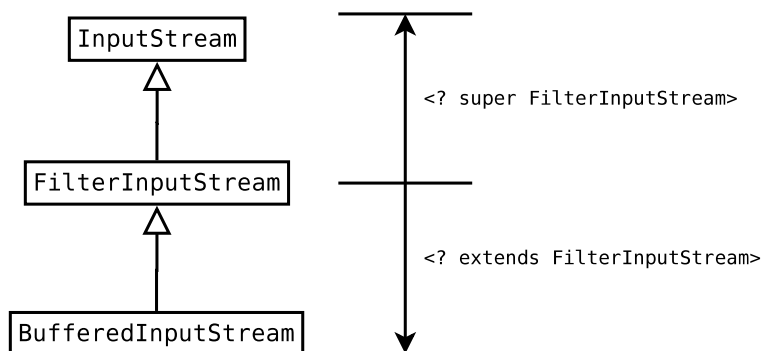


图 1.4: extends 和 super 确定了泛型的上下界

### 1.3.2.1 extends 声明了类型参数的上界

在尖括号中，可以使用 **extends** 明确限制类型参数是某个类的子类（包括本身）。比如 `<T extends Number>` 的意思是，类型参数 `T` 必须是 `Number` 类或者 `Number` 类的子类（`Integer`，`Double` 等），参见代码清单1.8。

代码清单 1.8: GenericBoxExtended.java

```

1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-23.
5  */
6 public class GenericBoxExtended<T extends Number> { // ❶
7     private T value;
8
9     public T getValue() {
10         return this.value;
11     }
12
13     public void setValue(T value) {
14         this.value = value;
15     }
16
17     public static void main(String[] args) {
18         GenericBoxExtended<Integer> boxInt = new GenericBoxExtended<Integer>();
19         boxInt.setValue(123); // ❷
20         System.out.println("456 + box' value = " + (456 + boxInt.getValue())); // ❸
21         GenericBoxExtended<Double> boxDouble = new GenericBoxExtended<Double>();
22         boxDouble.setValue(123.4);
23
24         //GenericBoxExtended<String> sbox = new GenericBoxExtended<String>(); // ❹
25     }
26 } //

```

❶ `T` 必须是 `Number` 类或者 `Number` 类的子类

- ❶ 这里利用了 Java 的自动装箱机制：整数 123 自动转换为 Integer 对象
- ❷ box 的 value 是一个整数，因此这里无须进行强制类型转换
- ❸ 由于 GenericBoxExtends 的类型参数限定了必须是 Number 的子类，因此这里使用 String 是非法的

我们再看一个经典的例子<sup>6</sup>：

代码清单 1.9: GenericExtendsTest.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-27.
5  */
6 public class GenericExtendsTest {
7     public static <T> int countGreaterThan(T[] anArray, T elem) {
8         int count = 0;
9         for (T e : anArray)
10             //if (e > elem) //❶
11                 ++count;
12         return count;
13     }
14 } //
```

❶ 语法错误，">" 只能用于简单数据类型，类型参数不允许多是简单数据类型，因此这里不能直接使用 ">"

在代码清单1.9中存在语法错误编译无法通过，原因是类型参数 T 不能为简单数据类型，因此不能使用关系运算符比较大小。知道了问题所在，也就容易想到解决方案了：使用 extends 限制类型参数必须是实现了 Comparable 接口的类即可，如此便可以借助于 Comparable 接口的 compareTo 方法比较两个对象的大小，如代码清单1.10所示。

代码清单 1.10: GenericExtendsComparableTest.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.Comparator;
4
5 /**
6  * Created by subaochen on 16-12-27.
7  */
8 public class GenericExtendsComparableTest {
9     public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
10         //❶
11         int count = 0;
12         for (T e : anArray)
13             if (e.compareTo(elem) > 0) // ❶
14                 ++count;
15         return count;
16     }
17 }
```

<sup>6</sup>借鉴自 Java Tutorial: <https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html>

```
16 } //
```

❶ 限定类型参数必须是实现了 `Comparable` 接口的类

❷ 借助于 `Comparable` 接口的 `compareTo` 方法判断两个对象的大小



如果 `extends` 后面是类名，则表示类型参数是指定类或者指定类的子类；如果 `extends` 后面是接口名，则表示类型参数是指定接口或者指定接口的实现类。

此外，`extends` 即可以用于泛型类的类型参数，也可以用于泛型方法的类型参数。



由于 `extends` 在这里可以表示接口的实现类，因此可以通过 `extends` 限定多个接口（可以包括一个类），比如：

```
1 public class A {}
2 public interface B {}
3 public interface C {}
4 public class D {}
5 public class Demo<T extends A & B & C> {}
6 public class Demo<T extends B & C & A> {} // 错误！ 类必须排在最前面
7 public class Demo<T extends A & D & B & C> {} // 错误！ 最多只能extends一个类，但是
    可以extends多个接口
```



关于对象的比较：如果两个对象要比较大小，则这两个对象必须实现 `Comparable` 接口。也就是说，Java 根据 `Comparable` 接口中的 `compareTo` 方法的返回值决定两个对象的比较结果。`Comparable` 接口的定义如下<sup>a</sup>：

```
1 public interface Comparable<T> {
2     public int compareTo(T o);
3 }
```

在 JDK API 文档中查看一下 `Integer`、`String` 等类即可以看出，它们都实现了 `Comparable` 接口。我们自定义的类只要实现了 `Comparable` 接口，即可以借助于 `compareTo` 方法方便的比较两个对象。

<sup>a</sup>参见 <http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>


### 1.3.2.2 `super` 声明了类型参数的下界

在尖括号中，可以使用 `super` 关键字明确限制方法参数为某个类的父类（包括本身）。比如 `<T super Box>`，则方法参数必须是 `Box` 类或者 `Box` 的父类，参见代码清单 1.11。

代码清单 1.11: GenericSuperTest.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-1-18.
5  */
6 public class GenericSuperTest {
7     public static void testSuper(GenericBox<? super Integer> box) {
8         box.setValue(1);
9         System.out.println("box.value = " + box.getValue());
10    }
11
12    public static void main(String[] args) {
13        testSuper(new GenericBox<Object>());
14    }
15 }
```

参见：Java Generics FAQ，`super` 不能用于类型参数，即 `super` 不能用于类的定义，比如这样的用法是错误的，Java 在语法上禁止了这种用法：



```
1 public class Box<T super Integer> {
2     private T data;
3     public Box(T data) { this.data = data; }
4     public T info() {
5         return data * 2;
6     }
7 }
```

这是因为所有的泛型都会在编辑阶段进行“类型擦除”，根据“子类型替换原则”，在类型参数中使用 `super` 是不安全的。

### 1.3.3 泛型方法使用泛型类的类型参数

在泛型方法中可以直接使用泛型类的类型参数作为返回值类型或者形式参数类型，当然，泛型方法也可以拥有自己独立的类型参数，参见代码清单1.12和代码清单1.13。

代码清单 1.12: Check.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * 本类演示了在泛型方法中直接使用泛型类的类型参数。
5  * Created by subaochen on 16-12-25.
6  */
7 public class Check<T> {
8     public boolean check(T value) {
9         return true;
10    }
```

```
10 }
11
12 public <U> U echo(U value) {
13     return value;
14 }
15 }
```

代码清单 1.13: CheckTest.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-25.
5  */
6 public class CheckTest {
7     public static void main(String[] args) {
8         Check<String> checkStr = new Check<String>();
9         System.out.println("check string: " + checkStr.check("a string"));
10        System.out.println("echo string: " + checkStr.echo("a string"));
11
12        Check<Integer> checkInt = new Check<Integer>();
13        System.out.println("check int: " + checkInt.check(100));
14        System.out.println("echo int: " + checkInt.echo(100));
15    }
16 }
```



不要混淆了泛型类的类型参数和泛型方法的类型参数。泛型类的类型参数在本类中有效，泛型方法的类型参数只在本方法中有效，因此泛型方法中可以直接使用泛型类的类型参数，但是泛型方法依然可以定义只属于本方法的类型参数列表，两者并不冲突。

## 1.4 使用泛型的注意事项

由于 Java 是通过类型擦除机制 [?] 实现泛型的，因此在具体使用泛型时应该特别注意以下几点：

- 类型参数的实际类型不能是简单的数据类型，比如 `int`，`long` 等，只能是类、接口、枚举类等。
- 类型参数并不能用来创建对象或是作为静态变量的类型，比如下面的示例代码，参见其中的注释：

```
1 class ClassTest<X extends Number, Y, Z> {
2     private X x;
```

```
3     private static Y y; //编译错误，不能用在静态变量中
4     public X getFirst() {
5         //正确用法
6         return x;
7     }
8     public void wrong() {
9         Z z = new Z(); //编译错误，不能创建对象
10    }
11 }
```

- 在使用带通配符的泛型类的时候，需要明确通配符所代表的一组类型的概念。由于具体的类型是未知的，很多操作是不允许的。
- 不要忽视编译器给出的警告信息。

## 1.5 类型擦除

Java 泛型这个特性是从 JDK 1.5 才开始加入的，因此为了兼容之前的版本，Java 泛型的实现采取了“伪泛型”的策略，即 Java 在语法上支持泛型，但是在编译阶段会进行所谓的“类型擦除”（Type Erasure），将所有的泛型表示（尖括号中的内容）都替换为具体的类型（其对应的原生态类型），就像完全没有泛型一样。理解类型擦除对于用好泛型是很有帮助的，尤其是一些看起来“疑难杂症”的问题，弄明白了类型擦除也就迎刃而解了。

泛型的类型擦除原则是：

- 消除类型参数声明，即删除 `<>` 及其包围的部分。
- 根据类型参数的上下界推断并替换所有的类型参数为原生态类型：如果类型参数是无限制通配符或没有上下界限定则替换为 `Object`，如果存在上下界限定则根据子类替换原则取类型参数的最左边限定类型（即父类）。
- 为了保证类型安全，必要时插入强制类型转换代码。
- 自动产生“桥接方法”以保证擦除类型后的代码仍然具有泛型的“多态性”。

### 1.5.1 擦除类定义中的类型参数

#### 1.5.1.1 无限制类型擦除

当类定义中的类型参数没有任何限制时，在类型擦除中直接被替换为 `Object`，即形如 `<T>` 和 `<?>` 的类型参数都被替换为 `Object`，参见图1.5。



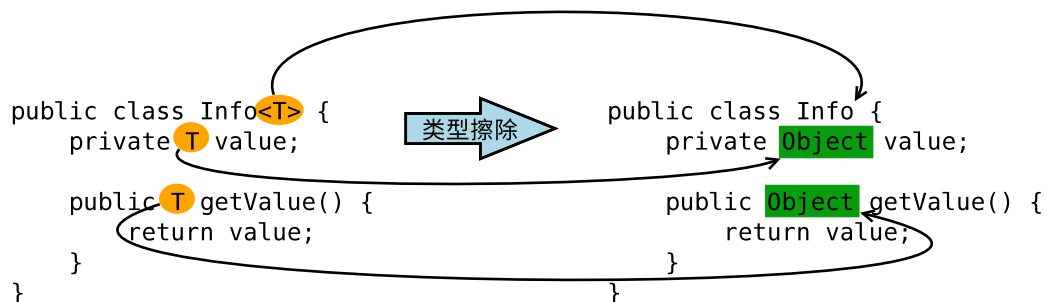


图 1.5: 擦除类定义中的类型参数

### 1.5.1.2 有限制类型擦除

当类定义中的类型参数存在限制（上下界）时，在类型擦除中替换为类型参数的上界或者下界，比如形如 `<T extends Number>` 和 `<? extends Number>` 的类型参数被替换为 `Number`，`<? super Number>` 被替换为 `Object`，参见图1.6。

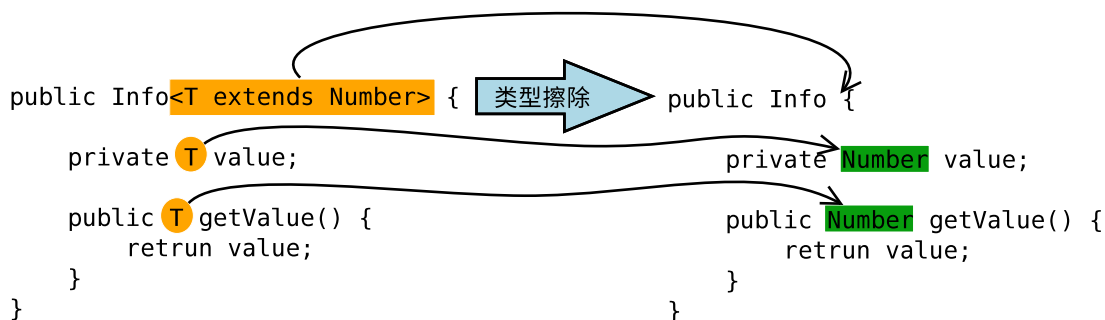


图 1.6: 擦除类定义中的有限制类型参数

### 1.5.2 擦除方法定义中的类型参数

擦除方法定义中的类型参数原则和擦除类定义中的类型参数是一样的，这里仅以擦除方法定义中的有限制类型参数为例，见图1.7。

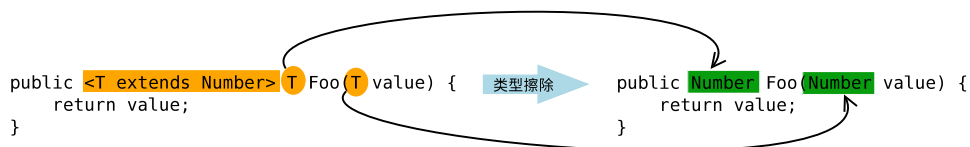


图 1.7: 擦除泛型方法中的类型参数

### 1.5.3 桥接方法和泛型的多态

考虑下面的代码：

```
1 public interface Info<T> {  
2     // just return var:-)  
3     T info(T var);  
4 }  
5 public class BridgeMethodTest implements Info<Integer> {  
6     @Override  
7     public Integer info(Integer var) {  
8         return var;  
9     }  
10 }
```

按照我们之前类型擦除的经验，在擦除类型后的代码应该是这个样子的：

```
1 public interface Info {  
2     // just return var  
3     Object info(Object var);  
4 }  
5  
6 public class BridgeMethodTest implements Info {  
7     @Override  
8     public Integer info(Integer var) {  
9         return var;  
10    }  
11 }
```

但是，明显可以看出，这样擦除类型后的代码在语法上是错误的：`BridgeMethodTest` 类中虽然存在一个 `info` 方法，但是和 `Info` 接口要求覆盖的 `info` 方法不一致：参数类型不一致。在这种情况下，Java 编译器会自动增加一个所谓的“桥接方法”（bridge method）来满足 Java 语法的要求，同时也保证了基于泛型的多态能够有效。我们反编译一下 `BridgeMethodTest.class` 文件可以看到 Java 编译器到底是如何做的：

```
$ javap BridgeMethodTest.class  
Compiled from "BridgeMethodTest.java"  
public class BridgeMethodTest implements Info<java.lang.Integer> {  
    public BridgeMethodTest();  
    public java.lang.Integer info(java.lang.Integer);  
    public java.lang.Object info(java.lang.Object);  
}
```

可以看出，Java 编译器在 `BridgeMethodTest` 中自动增加了两个方法：默认构造方法和参数为 `Object` 的 `info` 方法，参数为 `Object` 的 `info` 方法就是“桥接方法”。如理解“桥接”二字呢？我们进一步反编译 `BridgeMethodTest` 看一下：

```
1 // Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
2 // Jad home page: http://www.geocities.com/kpdus/jad.html
3 // Decompiler options: packimports(3)
4 // Source File Name: BridgeMethodTest.java
5
6
7 public class BridgeMethodTest
8     implements Info
9 {
10
11     public BridgeMethodTest()
12     {
13     }
14
15     public Integer info(Integer integer)
16     {
17         return integer;
18     }
19
20     public volatile Object info(Object obj)
21     {
22         return info((Integer)obj);
23     }
24 }
```

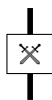
`info(Object)` 方法通过调用子类的 `info(Integer)` 方法搭起了父类和子类的桥梁，也就是说，`info(Object obj)` 这个方法起到了连接父类和子类的作用，使得 Java 的多态在泛型情况下依然有效。

当然，我们在使用基于泛型的多态时不必过多的考虑“桥接方法”，Java 编译器会帮我们打理好一切。

关于桥接方法的更多信息可以参考：JLS 的相关章节。

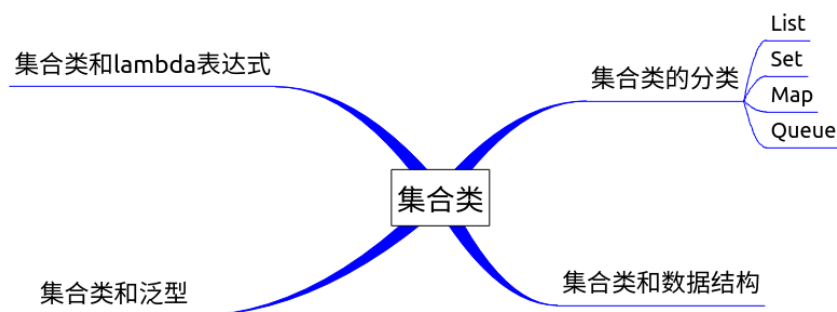
## 1.6 泛型的原生态类型

泛型的原生态类型（**raw type**）即擦除了泛型信息之后的真正类型。当我们定义一个泛型时，Java 会自动创建相应的原生态类型。在编译阶段，泛型会还原为原生态类型，类型参数被擦除并替换为其限定类型（无限定的类型使用 `Object` 替换）。



尽量避免使用泛型的原生态类型 [?, 第 23 条：请不要在新代码中使用原生态类型]。

## 第二章 集合类



### 2.1 集合的概念

“集合”是我们现实生活中经常遇到的数据表达方式，比如通讯录（记录了姓名、电话号码、住址等）、字典（记录了文字和其意义的对照关系）等等。为了更好的表达和处理集合，Java 提供了集合类框架（Collections Framework），包括以下三个部分：

- 一系列接口：虽然不同的集合内容（数据类型）可能不同，但是处理和操作集合的方式大致相似，因此集合类框架根据不同类型的集合定义了相应的接口以规范集合的操作。
- 接口的实现：上述接口的实现类。
- 算法：Java 的集合类框架提供了数据结构中常见的诸如排序、搜索等算法实现。

Java 的集合类框架带给我们的最明显好处就是，对于常见的数据结构，我们不再需要“重新发明轮子”，直接使用集合类框架即可，提高了编写代码的效率，也提高了代码的健壮性。z



在有的材料中把集合类叫做容器类 (Container)，道理是一样的：集合类框架的目的是处理批量的数据，就像将批量的数据放到一个容器中进行处理。基于此，本书可能在不同的场合交叉使用集合类和容器类两种说法。

## 2.2 集合类框架的接口

理解集合类框架的接口是用好集合类框架的关键,Java 的集合类框架接口如图2.1所示,主要分为两大类:

- 1. **Collection**: 就是通常的集合的概念。如果把 **Collection** 看做一个容器,那么 **Collection** 抽象的表达了如何看待容器中的元素: 如何存储、遍历、添加、删除等。
- 2. **Map**: 抽象的表达了“映射”的概念,即 **{key->value}** 这样的数据结构。

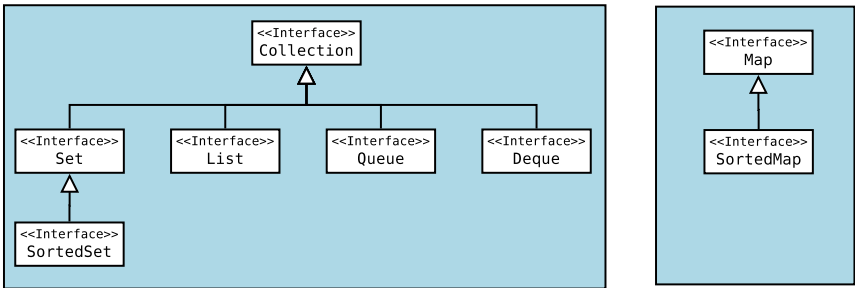


图 2.1: 集合类框架的接口

所有集合类框架中的接口和类都是泛型化的。实际上,Java 泛型的提出最初的目的就是为了增强集合类框架。也就是说,Collection 接口的定义是这样的:

```
1 public interface Collection<E> {...}
```

### 2.2.1 Collection 接口

根据图2.1, Collection 接口是所有集合类 (Map 除外) 的起点,即所有集合类均实现了 Collection 中所规定的方法。Collection 接口抽象的概括了我们可以对一个集合进行哪些操作,如表2.1所示。

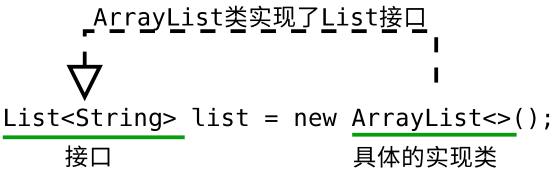
表 2.1: Collection 接口的方法

操作类型	方法	说明
基础操作	int size()	返回集合的元素个数
	boolean isEmpty()	判断集合是否为空
	boolean contains(Object element)	集合中是否包含对象 element
	boolean add(Object element)	添加对象 element 到集合中,成功则返回 true

	<code>boolean remove(Object element)</code>	从集合中删除对象 <code>element</code> ，成功则返回 <code>true</code>
	<code>Iterator&lt;E&gt; iterator()</code>	获得该集合的迭代器
批量操作	<code>boolean containsAll(Collection&lt;?&gt; c)</code>	集合 <code>c</code> 是否是当前集合的子集？
	<code>boolean addAll(Collection&lt;? extends E&gt; c)</code>	添加集合 <code>c</code> 的元素到当前集合（集合的并运算）
	<code>boolean removeAll(Collection&lt;?&gt; c)</code>	从当前集合删除集合 <code>c</code> 中的所有元素（集合的差运算）
	<code>boolean retainAll(Collection&lt;?&gt; c)</code>	从当前集合中删除不在集合 <code>c</code> 中的所有元素（集合的交运算）
	<code>void clear()</code>	清空集合
数组操作	<code>Object[] toArray()</code>	将集合转换为数组
	<code>&lt;T&gt; T[] toArray(T[] a)</code>	将集合转换为指定类型的数组
流式操作	<code>Stream&lt;E&gt; stream()</code>	获得集合的流式操作接口
	<code>Stream&lt;E&gt; parallelStream()</code>	获得支持并发的流式操作接口

我们将在下面的章节中结合更具体的集合类进一步看一下 `Collection` 接口的用法。

从宏观掌握 `Collection` 接口非常重要！在面向对象编程实践中，我们一致强调面向接口的编程，Java 的集合类框架在这一方面给了我们很好的示范。在使用 Java 的集合类框架时，我们也尽量要遵守面向接口编程的原则，如下图所示。



2.2.2 Set 接口

`Set` 接口是 `Collection` 的子接口，其表达的是数学上的集合概念：一组不重复的对象，即 `Set` 中的数据元素是不允许重复的，我们在下面往 `Set` 中添加和删除元素的时候可以看到如果往 `Set` 中添加重复的元素会发生什么。`Set` 的三个常见具体实现及其特点如表2.2所示。

由于 `Set` 不允许重复的对象，因此要特别注意 `Set` 的 `add` 和 `remove` 方法的返回值的意义，总结如下：

- 通过 `add` 方法添加一个数据元素到 `Set` 中的时候，如果 `Set` 中还没有该元素则

类名	说明
HashSet	数据元素存储到哈希表中，效率很高，但是不保证数据元素的顺序
TreeSet	数据元素存储到红黑树中，按照数据元素的大小排序，效率比 HashSet 低
LinkedHashSet	支持链表的 HashSet，数据元素按照其插入的顺序排序，效率比 HashSet 略低

表 2.2: Set 接口的具体实现类

添加并返回 `true`，否则返回 `false`。尤其是，Set 允许集合中包含 `null`，即 `null` 可以作为 Set 的一个数据元素存在。

- 通过 `remove` 方法从 Set 返回一个数据元素时，如果该数据元素存在则删除并返回 `true`，否则返回 `false`。

总而言之，`add` 和 `remove` 方法返回 `true`，如果 Set 被改变了的话，否则返回 `false`。

### 例 2.1. Set 的 `add`、`remove`、`size` 方法

代码设计 参见代码清单2.1。

代码清单 2.1: SetNullTest.java

```

1 package cn.edu.sdut.softlab;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 public class SetNullTest {
7
8     public static void main(String[] args) {
9         Set<String> set = new HashSet<>();
10        set.add("hello");
11        System.out.println("add null:" + set.add(null));
12        System.out.println("add null again:" + set.add(null));
13        System.out.println("set size:" + set.size());
14        System.out.println("remove null:" + set.remove(null));
15        System.out.println("set size:" + set.size());
16        System.out.println("remove null again:" + set.remove(null));
17        System.out.println("set size:" + set.size());
18    }
19 }

```

**运行结果** 程序运行结果如下：

```
add null:true
add null again:false
set size:2
remove null:true
set size:1
remove null again:false
set size:1
```

**代码说明** 可以看出，Set 允许在集合中包括 null 这个特殊的对象。add 和 remove 只有在集合切实被改变的时候才会返回 true，我们可以巧妙的利用这一点作为条件判断的依据，参见例2.3。

**例 2.2.** 统计给定字符串的不重复的单词及其个数

**代码设计** 参见代码清单2.2。

代码清单 2.2: FinDups.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 /**
7  * Created by subaochen on 17-1-21.
8  */
9 public class FinDups {
10     public static void main(String[] args) {
11         Set<String> s = new HashSet<String>();
12         for (String a : args)
13             s.add(a);
14         System.out.println(s.size() + " distinct words: " + s);
15     }
16 }
```

**运行结果** 本例要求运行时附带命令行参数，在 Idea 中可以在 Run 菜单选择“Edit Configurations...”填写命令行参数，如图2.2所示。

运行结果如下：

```
4 distinct words: [left, came, saw, I]
```



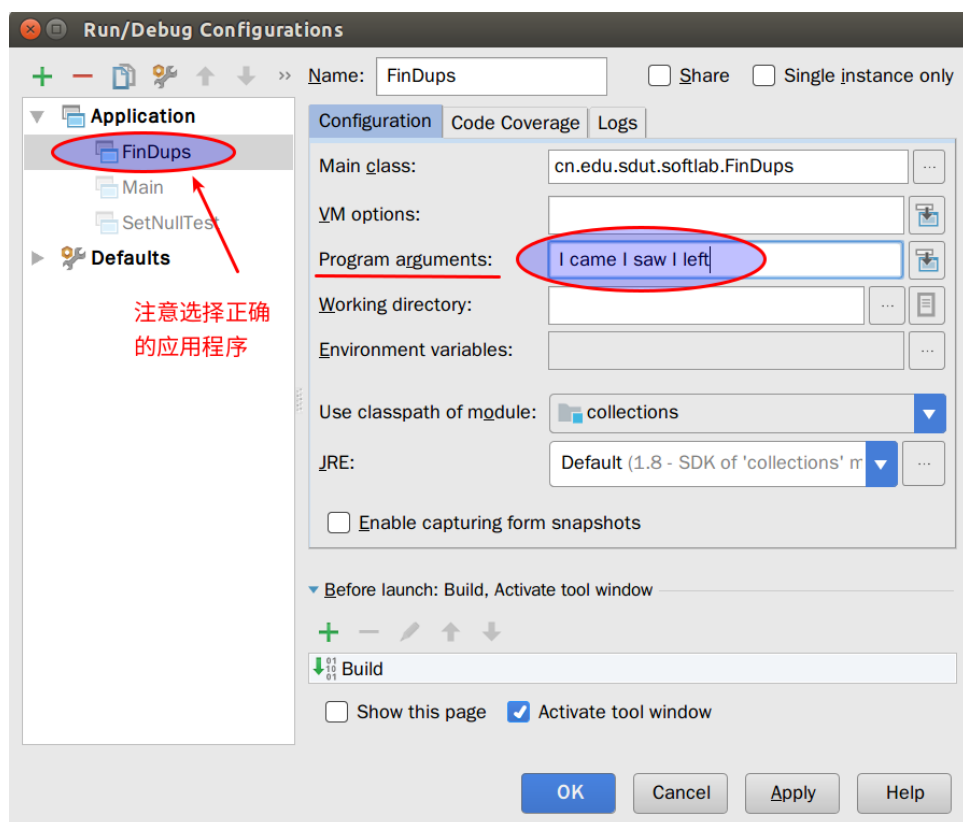


图 2.2: Idea 中设置应用程序的命令行参数

**代码说明** 本例中巧妙的利用了 `Set` 不允许重复元素的特点统计不重复的单词：直接遍历命令行参数数组并使用 `add` 方法添加即可，重复的数据是添加不到 `Set` 中的。另外可以看出，`System.out.println` 可以直接打印出 `Set` 集合的元素列表，请考虑一下 JDK 是如何做到这一点的<sup>1</sup>？

注意到，当我们反复执行本例时会发现输出的结果可能会不同，即打印出的单词顺序可能会有差异，这是因为 `HashSet` 并不保证元素的顺序。如果总是希望按照特定的顺序保存这些单词，可以将代码清单2.2中的 `HashSet` 简单替换为 `TreeSet` 或者 `LinkedHashSet`。请自行练习并认真体会 `HashSet` 和 `TreeSet`、`LinkedHashSet` 的区别。

### 例 2.3. 打印给定字符串的重复单词列表和唯一单词列表

**代码设计** 不同于例2.2，本例要求打印出命令行字符串中重复的单词和只出现了一次的单词列表，参见代码清单2.3

代码清单 2.3: FindDup2.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 /**
7  * Created by subaochen on 17-1-21.
8  */
9 public class FindDup2 {
10     public static void main(String[] args) {
11         Set<String> uniques = new HashSet<String>();
12         Set<String> dups = new HashSet<String>();
13
14         for (String a : args)
15             if (!uniques.add(a)) // 如果add方法返回false表明uniques中已经存在单词a，因此将a保存到dups中
16                 dups.add(a);
17
18         uniques.removeAll(dups); // 在for循环中，uniques已经获取了所有的单词，因此和dups的差即只出现一次的单词集合
19
20         System.out.println("Unique words: " + uniques);
21         System.out.println("Duplicate words: " + dups);
22     }
23 }
```

<sup>1</sup>提示：可以查阅 `openjdk` 源代码的 `AbstractCollection` 类，其中实现了 `toString` 方法来遍历集合元素并构造一个描述集合元素的字符串。

**运行结果** 参照例2.2，我们设置应用程序的命令行参数为 `I came I saw I left`，则执行结果为：

```
Unique words: [left, came, saw]
Duplicate words: [I]
```

**代码分析** 本例的重点是：

- 巧妙的利用了 `Set` 的 `add` 方法的特点构造了判断条件：如果要添加的元素已经存在于集合中则返回 `false`。
- 使用了两个 `Set` 来保存单词列表，`uniques` 在循环中首先保存所有的单词，然后和保存重复单词的集合 `dups` 做“差运算”即可获得只出现一次的单词列表。

假设我们有一个集合 `c`，希望通过这个集合创建一个去除重复元素的新集合，可以借助于 `Set` 接口的特性使用一行代码来完成<sup>a</sup>：

```
1 Collection<Type> noDups = new HashSet<Type>(c);
```



使用 JDK 8 的流式处理可以这样实现：

```
1 c.stream().collect(Collector.toSet());
```

<sup>a</sup>参见

### 2.2.3 List 接口

`List` 接口是 `Collection` 的子接口，表达了一个有序的数据集合，其突出特点为：

- `List` 中的数据是根据其加入（调用 `add` 或者 `addAll` 方法）的顺序排序的。
- `List` 中的数据允许重复。

`List` 中的数据是有序的，因此很自然的我们可以按照数据的序号（`List` 的序号遵循 C 语言的习惯，也是从 0 开始的）来操作数据，因此 `List` 接口除了具有 `Collection` 接口的所有方法外，还定义了如表2.3所示的额外方法。

`List` 接口最常见的三个实现类是 `ArrayList`、`LinkedList` 和 `Vector`，由于其内部实现不同，在不同的场合下的性能差异比较大，表2.4总结了 `ArrayList`、`LinkedList`、`Vector` 的异同。

#### 例 2.4. 基本的 List 操作示例

操作类型	方法名称	说明
位置操作	E get(int index)	获得序号为 i 的对象
	E set(int index, E element)	使用 element 替换序号为 i 处的对象
	void add(int index, E element)	在序号 i 处插入对象 element 并依次后移受影响的其他对象
	boolean addAll(int index, Collection<? extends E> c)	在序号 i 处插入集合 c 并依次后移受影响的其他对象
	E remove(int index)	删除序号 i 处的对象，并依次前移对象填补空缺
搜索	int indexOf(Object o)	获得对象 o 在 List 中第一次出现的序号，如果对象不在 List 中则返回-1
	int lastIndexOf(Object o)	获得对象 o 在 List 中最后出现的序号，如果对象不在 List 中则返回-1
	List<E> subList(int fromIndex, int toIndex)	获得序号 fromIndex（包含）到 toIndex（不包含）之间的数据组成的 List
迭代	ListIterator<E> listIterator(int index)	获得一个从起始序号 index 开始的 ListIterator
排序	void sort(Comparator<? super E> c)	根据给定的 comparator 排序 List

表 2.3: List 定义的额外方法

类名	特点	使用场合
ArrayList	ArrayList 是基于动态数组（可改变大小）的数据结构，随着数据元素的不断加入和删除，ArrayList 的大小会不断调整。ArrayList 不是线程安全的。	随机访问频繁时，ArrayList 有速度优势。在大多数情况下，应该优先选用 ArrayList
LinkedList	LinkedList 是基于双向链表的数据结构。LinkedList 不是线程安全的。	频繁添加和删除操作时，LinkedList 有速度优势
Vector	相当于线程安全的 ArrayList	Vector 的执行效率不高，只在并发编程中考虑使用 Vector

表 2.4: List 的具体实现类

代码设计 参见代码清单2.4。

代码清单 2.4: ListBasic.java

```

1 package cn.edu.sdut.softlab;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * Created by subaochen on 17-1-22.
8  */
9 public class ListBasic {
10     public static void main(String[] args) {
11         List<Integer> list1 = new ArrayList<>();
12         List<Integer> list2 = new ArrayList<>();
13
14         for(int i = 0; i < 10; i++) {
15             list1.add(i);
16             list2.add(i + 20);
17         }
18
19         System.out.println("list1 elements:" + list1);
20         System.out.println("list2 elements:" + list2);
21         list1.addAll(list2);
22         System.out.println("list1 + list2 elements:" + list1);
23
24         for(int i = 0; i < 20; i = i + 3) {
25             System.out.println("read from list1, index = " + i + ", value = " + list1.get(i));
26         }
27

```

```
28     for(int i = 0; i < 20; i = i + 5) {
29         System.out.println("will replace the " + i + " elements with: " + list1.get(i)
30             ) * 5);
31         list1.set(i, list1.get(i) * 5);
32     }
33     System.out.println("list1 elements now:" + list1);
34
35     // 在第10个元素后插入一个新元素33
36     // @TODO 此种情况下使用ArrayList的效率没有LinkedList高, 尝试使用LinkedList替代
37     list1.add(10,33);
38     System.out.println("list1 elements after added:" + list1);
39
40     // 删除第11个元素
41     list1.remove(11);
42     System.out.println("list1 elements after removed:" + list1);
43 }
```

**运行结果** 运行结果如下:

```
list1 elements:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list2 elements:[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
list1 + list2 elements:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29]
read from list1, index = 0, value = 0
read from list1, index = 3, value = 3
read from list1, index = 6, value = 6
read from list1, index = 9, value = 9
read from list1, index = 12, value = 22
read from list1, index = 15, value = 25
read from list1, index = 18, value = 28
will replace the 0 elements with: 0
will replace the 5 elements with: 25
will replace the 10 elements with: 100
will replace the 15 elements with: 125
list1 elements now:[0, 1, 2, 3, 4, 25, 6, 7, 8, 9, 100, 21, 22, 23, 24,
125, 26, 27, 28, 29]
list1 elements after added:[0, 1, 2, 3, 4, 25, 6, 7, 8, 9, 33, 100, 21,
22, 23, 24, 125, 26, 27, 28, 29]
list1 elements after removed:[0, 1, 2, 3, 4, 25, 6, 7, 8, 9, 33, 21, 22,
23, 24, 125, 26, 27, 28, 29]
```

**代码说明** 本例综合演示了 List 的基本操作，包括 get, set, add, remove, addAll 等方法。

ArrayList 的默认构造方法会创建一个初始大小为 10 的数组<sup>a</sup>，如果预计到要使用到更多的数据，最好不要使用使用 ArrayList 默认的构造方法，而是使用 ArrayList(int initialCapacity) 创建一个合适大小的数组，以减小不断扩大数组的无谓开销。比如下面的代码创建一个初始大小为 100 的 ArrayList 对象：



```
1 List<String> list = new ArrayList<>(100);
```

<sup>a</sup>参见 openjdk 的源代码: jdk/src/java.base/share/classes/java/util/ArrayList.java

### 例 2.5. 将命令行参数随机乱序输出到屏幕

**代码设计** 参见代码清单2.5。

代码清单 2.5: Shuffle.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.List;
6
7 /**
8  * Created by subaochen on 17-1-23.
9  * 本例来自java tutorial
10 */
11 public class Shuffle {
12     public static void main(String[] args) {
13         List<String> list = new ArrayList<>();
14         // 下面的for循环可以使用Arrays.asList替代，更简洁和高效
15         // list = Arrays.asList(args);
16         for (String a : args)
17             list.add(a);
18         Collections.shuffle(list);
19         System.out.println(list);
20     }
21 }
```

**运行结果** 假设命令行参数为“I came I saw I left”，则运行后可能的结果为（每次运行结果可能不同）：

[I, left, saw, I, came, I]

**代码说明** 本例借助于 `Collections` 类的 `shuffle` 方法乱序给定的 `List`。如果我们自己写一个 `shuffle` 方法呢？参考下面的代码：

```
1 public static <E> void swap(List<E> a, int i, int j) {
2     E tmp = a.get(i);
3     a.set(i, a.get(j));
4     a.set(j, tmp);
5 }
6 public static void shuffle(List<?> list) {
7     Random rnd = new Random();
8     for (int i = list.size(); i > 1; i--)
9         swap(list, i - 1, rnd.nextInt(i));
10 }
```

`Collections`<sup>a</sup>是一个集合类的辅助工具类，其中定义了一系列类方法，比较常用的有：

- `shuffle`：乱序给定的 `List`
- `reverse`：逆序给定的 `List`
- `swap`：交换给定 `List` 的指定元素
- `fill`：用给定的对象填充 `List`
- `max/min`：返回给定 `Collection` 中元素的最大值/最小值



`Arrays`<sup>b</sup>是另外一个常见的工具类，顾名思义，`Arrays` 提供了和数组相关的一些类方法，常见的有：

- `asList`：将给定的数组转化为一个 `List`
- `sort`：排序给定的数组
- `fill`：填充给定的数组
- `copyOf`：将给定的数组截断后复制为一个新数组

<sup>a</sup>详情参见：<http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

<sup>b</sup>详情参见：<http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

### 例 2.6. `subList` 的用法示例

`subList` 是一个很有特色的方法，其特点为：

- 半开半闭：`subList` 包含起点但是不包含终点位置。
- 非独立存在的 `List`，而是原 `List` 的一个视图，即对 `subList` 的变化会马上反映到



原 List 上面。这很像数据库中“视图”(View)的概念和用法。

代码设计 参见代码清单2.6。

代码清单 2.6: SubListTest.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * Created by subaochen on 17-1-23.
8  */
9 public class SubListTest {
10     public static void main(String args[]) {
11         List<Integer> list = new ArrayList<>();
12
13         list.add(1);
14         list.add(2);
15         list.add(3);
16         list.add(4);
17         list.add(5);
18         list.add(6);
19         list.add(7);
20
21         //
22         //
23         List<Integer> subList = list.subList(1, 6); // ❶
24         System.out.println(subList);
25
26         subList.add(10); // ❶
27         System.out.println(list);
28
29         list.add(3, 20); // ❷
30         //System.out.println(subList);
31
32         // 下面的代码抛出异常：起点不能大于终点
33         System.out.println("----");
34         list.subList(6, 5);
35     }
36 } //
```

❶ 注意到 subList 的起点是从 0 开始计算的，因此 (1,6) 应该打印出 2-6。

❶ 操作 subList 后，可以看到在原 list 的数据也发生了改变，也就是说，subList 是 list 的一个视图。

❷ 但是，如果修改了原 list，则 subList 就变得不可用，除非重新执行 subList 操作。也就是说，subList 作为 List 的视图是一个单向的关系，修改 subList 的内容可以反映到 list 中，但是修改 list 无法反映到 subList 中。

运行结果 程序运行结果如下：

```
[2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 10, 7]
----
Exception in thread "main" java.lang.IllegalArgumentException: fromIndex(6) > toIndex(5)
at java.util.ArrayList.subListRangeCheck(ArrayList.java:1006)
at java.util.ArrayList.subList(ArrayList.java:996)
at cn.edu.sdut.softlab.SubListTest.main(SubListTest.java:34)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
.....
at java.lang.reflect.Method.invoke(Method.java:498)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)
```

**代码说明** `subList` 的意义比较明确，但是如果要修改 `subList` 的内容或者原 `List` 的内容还是要特别小心，要注意到 `subList` 作为原 `List` 的一个视图是单向的映射关系，参见代码清单2.6中的注解。

## 2.2.4 Queue 接口

`Queue` 接口描述了《数据结构》中“队列”的概念，因此本节内容要求读者了解一定的队列基础知识。简单的说，队列就像一个两端开口的管道，数据总是一个一个从左端进入，右端取出，即所谓的“先进先出”（FIFO: First In First Out）结构<sup>2</sup>。`Queue` 反映了生活中的“排队”现象，比如火车站排队买票，食堂学生排队打饭，打印机的打印任务缓冲队列等等，即“先来先服务”模式，如图2.3所示，

图2.3也列出了 `Queue` 的主要方法。`Queue` 的每个方法都有两种形式：

1. 如果操作失败则抛出异常，因此后续的操作将无法进行；
2. 如果操作失败返回特定的值，比如 `null` 或者 `false`，可以根据这个返回值决定是否进行后续的操作。

具体来说，`Queue` 的方法如表2.5所示<sup>3</sup>。

`Queue` 接口的主要实现类有：

- `LinkedList`

---

<sup>2</sup>Java `Queue` 接口的某些实现类也提供了特殊形式的队列，这些队列形态不一定是 FIFO 的，比如 `PriorityQueue` 根据数据元素的值而非入队顺序排序。

<sup>3</sup>根据方法的第一个字母可以想到一个简单的策略记忆其差别：`are exception`, `pop failure`（队列皆异常，弹出都失败）。

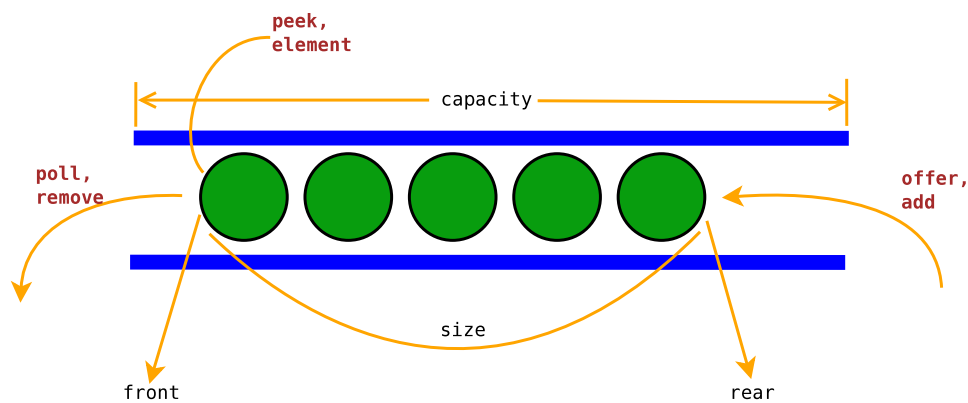


图 2.3: Queue 示意图

操作类型	抛出异常	返回特定值	说明
插入	<code>add(e)</code>	<code>offer(e)</code>	在队尾插入元素，如果队列已满，则 <code>add</code> 抛出 <code>IllegalStateException</code> ，而 <code>offer</code> 返回 <code>false</code> 。
删除	<code>remove()</code>	<code>poll()</code>	从队头删除元素并返回被删除的元素。如果队列已经为空， <code>remove</code> 抛出 <code>NoSuchElementException</code> ，而 <code>poll</code> 返回 <code>null</code> 。
检查	<code>element()</code>	<code>peek()</code>	检查队头元素（不删除），如果队列已经为空， <code>element</code> 抛出 <code>NoSuchElementException</code> ，而 <code>peek</code> 返回 <code>null</code> 。

表 2.5: Queue 的主要方法

- PriorityQueue

- ArrayDeque

**例 2.7.** Queue 综合示例：设计一个倒计时计数器。

**代码设计** 参见代码清单2.7。

代码清单 2.7: Countdown.java

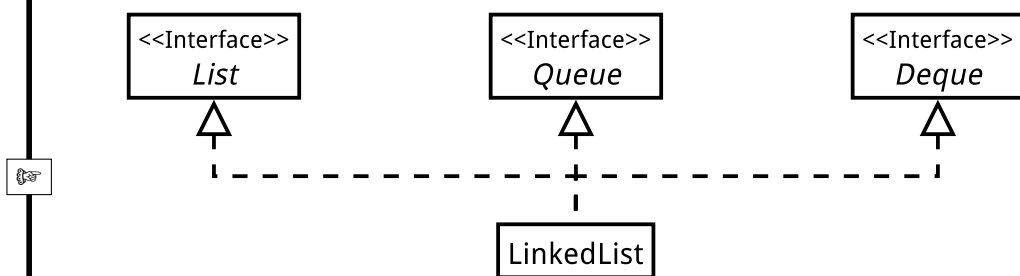
```
1 package cn.edu.sdut.softlab;
2
3 import java.util.LinkedList;
4 import java.util.Queue;
5
6 /**
7  * Created by subaochen on 17-1-26.
8  */
9 public class Countdown {
10     public static void main(String[] args) throws InterruptedException {
11         int time = Integer.parseInt(args[0]);
12         Queue<Integer> queue = new LinkedList<Integer>();
13
14         for (int i = time; i >= 0; i--)
15             queue.add(i);
16
17         while (!queue.isEmpty()) {
18             System.out.println(queue.remove());
19             Thread.sleep(1000);
20         }
21     }
22 }
```

**运行结果** 首先设置运行时的命令行参数??，比如设置为 10，运行结果如下：

10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0

**代码分析** 对于这样的简单场景，我们当然不一定必须使用 `Queue` 来实现<sup>4</sup>。为了实现倒计时，我们首先构造了一个 `Queue` 并加入了一组整数，然后每隔一定时间（这里是 1 秒）依次取出这组整数即可。

`LinkedList` 实现了两个接口：`Queue` 和 `List`，如下图所示：



因此，`LinkedList` 是一个应用很广泛的实用类，值得我们认真研究和扎实掌握起来。

**Deque** 是一个双向 `Queue` 接口，即允许在两端进行读写的 `Queue`，请读者自行参考相关资料了解详情。`LinkedList` 也实现了 `Deque` 接口。

### 2.2.5 Map 接口

`Map` 接口表达了数据映射的概念，一个 `Map` 对象中包含两个集合：`Key` 集合和 `Value` 集合。`Map` 对象中 `Key` 集合和 `Value` 集合的关系非常像“连连看”：

<sup>4</sup>作为练习，这里请考虑还有什么方法实现这个功能？比如，可以使用多线程实现吗？

Map<K, V> 接口的操作方法如表2.6所示。

操作类型	名称	说明
基本操作	V put(K key, V value)	将 (key,value) 对加入到当前 map 对象
	V get(Object key)	根据 key 从当前 map 对象获得相应的 value
	V remove(Object key)	根据 key 从当前 map 删除一个映射, 返回被删除的 value
	boolean containsKey(Object key)	检查当前 map 是否包含给定的 key
	boolean containsValue(Object value)	检查当前 map 是否包含给定的 value
	int size()	返回当前 map 的大小 (key-value 对的个数)
	boolean isEmpty()	检查当前 map 是否为空
批量操作	void clear()	清除 Map
	void putAll(Map<? extends K,? extends V> m)	将 m 添加到给定的 map 对象
集合视图	Set<K> keySet()	当前 map 所有 key 的 Set 集合
	Collection<V> values()	当前 map 所有 value 的 Collection 集合
	Set<Map.Entry<K,V>> entrySet()	当前 map 的所有 key-value 对的 Set 集合

表 2.6: Map 接口的方法

Map 接口的三个常见实现类是 HashMap、TreeMap、LinkedHashMap。从名字上可以看出, 这三个实现类和 Set 接口的三个实现类 HashSet、TreeSet、LinkedHashSet 非常相似, 实际上也是这样, 我们通过下面的例子可以看出。

2.2.5.1 Map 的基本操作

例 2.8. 统计命令行参数中字符串的出现频率

代码设计 参见代码清单2.8。

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashMap;
4 import java.util.LinkedHashMap;
5 import java.util.Map;
6 import java.util.TreeMap;
7
8 /**
9  * Created by subaochen on 17-1-27.
10  */
11 public class WordFreq {
12     public static void main(String[] args) {
13         Map<String, Integer> m = new HashMap<>();
14         // 根据单词的自然顺序排序
15         //Map<String, Integer> m = new TreeMap<>();
16         // 根据添加的顺序排序
17         //Map<String, Integer> m = new LinkedHashMap<>();
18
19         // 从命令行解析字符串并保存到map中
20         for (String a : args) {
21             Integer freq = m.get(a); // 根据Key获得指向的对象
22             m.put(a, (freq == null) ? 1 : freq + 1); // 根据key保存指向的对象 (value)。如果对
                // 象已经存在则覆盖
23         }
24
25         System.out.println(m.size() + " distinct words:");
26         System.out.println(m);
27     }
28 }
```

**运行结果** 假设命令行参数为 “If it is to be it is up to me”，则运行结果如下：

```
7 distinct words:
{be=1, me=1, is=2, it=2, to=2, up=1, If=1}
```

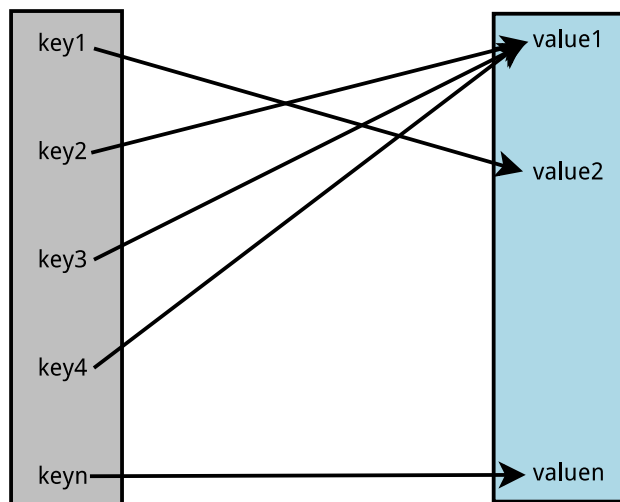
**代码说明** HashMap 中保存的数据并不保证顺序，也就是说，HashMap 中的 key->value 对不是按照加入的顺序或者值的大小来排序的。在例2.8中，如果希望最终的统计结果是按照单词出现的顺序排序的，则可以把 HashMap 换成 LinkedHashMap；如果希望最终的统计结果是按照单词的自然顺序排序，则可以把 HashMap 换成 TreeMap，读者可以自行尝试并运行查看结果。



Map 中的 Key 和 Value 的对应关系是唯一的，即 Map 中不允许存在重复的 Key->Value 关系，这和 Set 接口有相似之处。我们可以反过来考虑这个问题：假设 Map 允许存在重复的 Key->Value 关系，那么 get(Key) 方法将返回不止

一个结果，这在现实中也是不合理的：比如我们查字典，给定一个单词却告诉我们在第 100 页和第 200 页都存在这个单词，我们该相信哪个呢？

注意到，所谓的“Key 和 Value 的对因关系是唯一的”，即 Key 集合中的元素不重复，或者说 Key 集合是一个 Set，而 Value 集合是允许重复的，即允许存在下图所示的情形：



### 2.2.5.2 批量操作

Map 的批量操作主要是 `putAll` 方法的使用。和 `Collection` 的 `addAll` 方法很类似，Map 的 `putAll` 方法可以将一个 map 的所有内容添加到当前 map 中。

**例 2.9.** 比如设计一个参考文献管理的应用程序，每个参考文献由索引和标题两部分组成。编写程序，把来自两个参考文献库的内容合并。

**代码设计** 参见代码清单2.9。

代码清单 2.9: RefManager.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * Created by subaochen on 17-1-30.
8  */
9 public class RefManager {
10     public static void main(String[] args) {
11         Map<String, String> refl = new HashMap<>();
12         refl.put("java-effective", "Java Effective(2nd edition)");
13         refl.put("tij", "Thinking in Java(3th edition)");
```



```
14     ref1.put("java-tutorial", "Oracle Java Torial");
15     System.out.println("ref1 size:" + ref1.size() + "," + ref1);
16
17     Map<String, String> ref2 = new HashMap<>();
18     ref2.put("generics-collections", "Java Generics and Collections");
19     ref2.put("tij", "Thinking in Java(4th edition)");
20     System.out.println("ref2 size:" + ref2.size() + "," + ref2);
21
22     ref1.putAll(ref2); // 合并ref2到ref1中
23     System.out.println("after putAll, ref1 size:" + ref1.size() + "," + ref1);
24 }
25 }
```

**运行结果** 程序运行结果如下：

```
ref1 size:3,{java-tutorial=Oracle Java Torial, tij=Thinking in Java(3th edi-
tion), java-effective=Java Effective(2nd edition)}
ref2 size:2,{tij=Thinking in Java(4th edition, generics-collections=Java
Generics and Collections}
after putAll, ref1 size:4,{java-tutorial=Oracle Java Torial, tij=Thinking in
Java(4th edition, generics-collections=Java Generics and Collections, java-
effective=Java Effective(2nd edition)}
```

**代码说明** 可以看出，putAll 方法合并了两个 Map，合并的结果是，相同 key 的 Entry 被覆盖掉了，如例子中 key=tij 的项。

### 2.2.5.3 集合视图

所谓“集合视图”是指从集合的角度看 Map，可以把 Map 分为如下的 3 种集合：

- keySet：所有 key 的集合，是一个 Set，因此 key 是不允许重复的。
- values：所有 value 的集合，是一个 Colleciton，允许重复。
- entrySet：所有 key-value 对的集合，是一个 Set，其中的每一个 key-value 对（即 Set 中的每个元素是一个 Map.Entry）通过 Map 的内部接口 Map.Entry 来表达。

当需要把 Map 作为集合来处理的时候，这三种集合视图就非常有用了。

**例 2.10.** 遍历 Map 的 2 种方法

代码设计 参见代码清单2.10。

代码清单 2.10: MapTraverse.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * Created by subaochen on 17-1-30.
8  */
9 public class MapTraverse {
10     public static void main(String[] args) {
11         Map<Integer, String> map = new HashMap<>();
12         map.put(1, "one");
13         map.put(2, "two");
14         map.put(3, "three");
15         map.put(4, "four");
16
17         for(int key : map.keySet()) {
18             System.out.println(key + "->" + map.get(key));
19         }
20
21         // 通过Map.Entry可以在遍历Map时修改value的值
22         for(Map.Entry<Integer, String> entry : map.entrySet()) {
23             entry.setValue(entry.getValue() + " apple");
24             System.out.println(entry.getKey() + "->" + entry.getValue());
25         }
26     }
27 }
```

运行结果 运行结果如下：

```
1->one
2->two
3->three
4->four
1->one apple
2->two apple
3->three apple
4->four apple
```

代码分析 遍历 Map 的两种常见方法是根据 keySet 遍历和根据 Map.EntrySet 遍历。根据 Map.EntrySet 遍历的额外好处是在遍历中可以检查和修改 value 的值，这

是在遍历 Map 时唯一修改 value 值的方式<sup>5</sup>。

2.2.6 Iterator 接口

Iterator（迭代器）是一个标准的通用接口，用于遍历一个集合（Collection），因此 List、Set、Queue 都可以使用 Iterator 进行遍历操作。Iterator 接口的主要方法如表2.7所示。

方法	说明
boolean hasNext()	判断游标右边是否还有可读的元素
E next()	读取游标右边的元素，并将游标移动到下一个位置
void remove()	删除游标左边的元素。在执行完 next 之后，该操作只能执行一次

表 2.7: Iterator 的主要方法

正确使用 Iterator 的关键是如何理解 Iterator 的当前“指针”，如图2.4所示。

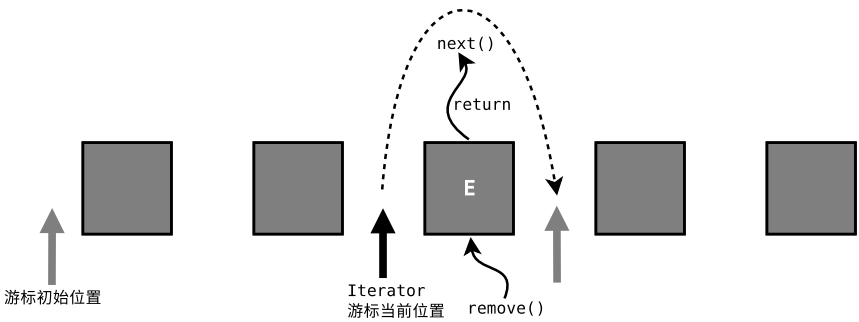


图 2.4: Iterator 示意图

例 2.11. 使用 Iterator 遍历 Set

代码设计 参见代码清单2.11。

代码清单 2.11: SetIteratorTest.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashSet;
4 import java.util.Iterator;
5 import java.util.Set;
```

<sup>5</sup>在遍历时顺便删除 Map 中某个 key-value 对的办法参见例2.13。

```
6
7 /**
8  * Created by subaochen on 17-1-30.
9  */
10 public class SetIteratorTest {
11     public static void main(String[] args) {
12         Set<String> set = new HashSet<>();
13         set.add("java");
14         set.add("language");
15         set.add("is");
16         set.add("not");
17         set.add("good");
18
19         // 使用Iterator遍历set, 并将包含not的元素删除
20         for (Iterator<String> it = set.iterator(); it.hasNext(); ) { //❶
21             String item = it.next(); // ❶
22             if (item.contains("not")) {
23                 it.remove(); // ❷
24             }
25         }
26
27         System.out.println(set);
28     }
29 } //
```

❶ 使用 for 循环, 注意初始化 Iterator 对象 it 的方法; 也需要注意 for 循环的第三个表达式是空的

❶ 读取游标右边的元素, 并将游标移动到下一个位置

❷ 删除游标左边的元素: 注意, 通常此时游标已经通过 next 移动到了下一个位置!

**运行结果** 运行结果如下:

```
[java, language, is, good]
```

**代码说明** 所有需要注意的事项都在代码注释中。

**例 2.12.** 使用 Iterator 遍历 List

**代码设计** 参见代码清单2.12。

代码清单 2.12: ListIteratorTest.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.Iterator;
6 import java.util.List;
```

```
7
8 /**
9  * Created by subaochen on 17-1-30.
10 */
11 public class ListIteratorTest {
12     public static void main(String[] args) {
13         List<String> list = new ArrayList<>();
14         list.add("java");
15         list.add("language");
16         list.add("is");
17         list.add("not");
18         list.add("funny");
19
20         Iterator<String> it = list.iterator();
21         while(it.hasNext()) {
22             if(it.next().contains("not")) it.remove();
23         }
24
25         System.out.println(list);
26     }
27 }
```

**运行结果** 运行结果如下：

[java, language, is, funny]

**代码说明** 本例除了使用 while 循环替代了 for 循环外，和例2.11几乎是一样的。

在构造 List 对象时经常使用 Arrays.asList 方法，简洁方便，比如：

```
1 List<String> list = Arrays.asList(new String[]{"abc", "def"});
```

但是这种方法也有一个“致命”的弱点，即这种方法构造的 list 是基于数组的，其大小是固定的，既不能删除 (remove) 元素，也不能添加 (add) 元素。比如下面的代码会抛出 UnsupportedOperationException：



```
1 List<String> list = Arrays.asList(new String[]{"abc", "def"});
2 for(Iterator it = list.iterator(); it.hasNext();) {
3     if(it.next().contains("not")) it.remove();
4 }
```

因此，在可能改变（添加或者删除）List 中的元素时，不能使用 Arrays 工具类初始化 List 对象，而应该使用 ArrayList 等创建 List 对象，如例2.12所示。

**练习 2.1.** 下面这段代码演示了 ListIterator 的用法，请说明其功能是什么？

```
1 public static <E> void replace(List<E> list, E val, E newVal) {
2     for (ListIterator<E> it = list.listIterator(); it.hasNext(); )
3         if (val == null ? it.next() == null : val.equals(it.next()))
4             it.set(newVal);
5 }
```



Iterator 是一个很好的概念设计，但是随着 Java 中增加了 for-each 语法，使用 for-each 遍历一个集合通常更方便，因此除非在遍历集合时需要删除集合元素，通常应该使用 for-each 来遍历一个集合。

**例 2.13.** 使用 Iterator 遍历 Map 并有选择的删除 Map 中的元素

**代码设计** 参见代码清单2.13。

代码清单 2.13: MapIterator.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.HashMap;
4 import java.util.Iterator;
5 import java.util.Map;
6
7 /**
8  * Created by subaochen on 17-1-30.
9  */
10 public class MapIterator {
11     public static void main(String[] args) {
12         Map<Integer, String> map = new HashMap<>();
13         map.put(1, "one");
14         map.put(2, "two");
15         map.put(3, "three");
16         map.put(4, "four");
17
18         for (Iterator<Integer> it = map.keySet().iterator(); it.hasNext(); ) {
19             if (it.next() % 2 == 0) {
20                 it.remove();
21             }
22         }
23
24         System.out.println(map);
25     }
26 }
```

**运行结果** 运行结果如下：

```
{1=one, 3=three}
```

**代码说明** Iterator 接口是遍历时唯一安全删除 Map 中元素的方法。如果需要在遍历时修改元素，唯一可行的方法是遍历 Map.EntrySet 并利用 EntrySet 提供的 setValue 方法。

### 2.2.7 Collection 的 addAll 方法

Collection 的 addAll 方法定义如下：

```
1 interface Collection<E> {  
2     ...  
3     public boolean addAll(Collection<? extends E> c);  
4     ...  
5 }
```

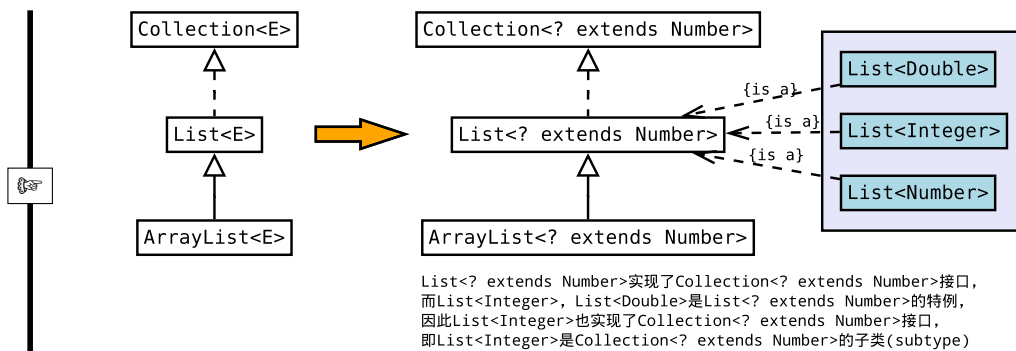
也就是说，对于一个类型为 E 的集合（即集合中的元素数据类型为 E），可以将另外一个类型为 E 的一个集合添加到这个集合中去，这是合理的和容易理解的。addAll 方法的参数使用了泛型技术，即任何类型为 E 或者 E 的子类的集合，都可以添加到这个集合中。

```
1 List<Number> nums = new ArrayList<Number>();  
2 List<Integer> ints = Arrays.asList(1, 2);  
3 List<Double> dbls = Arrays.asList(3.45, 1.78);  
4 nums.addAll(ints);  
5 nums.addAll(dbls);  
6 assert nums.toString().equals("[1, 2, 3.45, 1.78]");
```

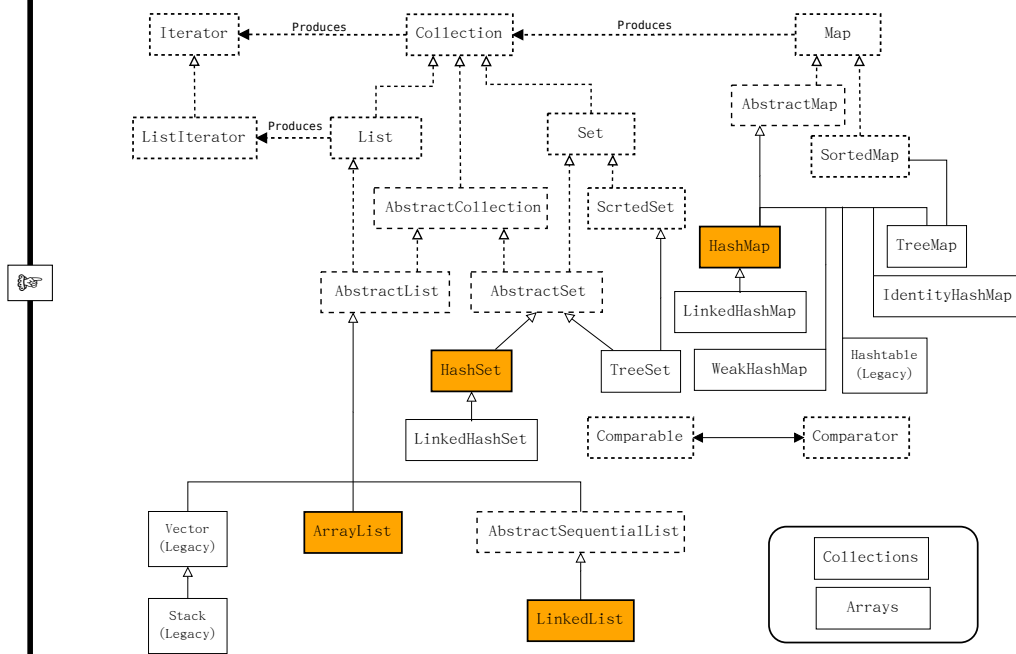
我们重点看一下 nums.addAll(ints)。可以从两个层面理解这一调用：第一，nums 是一个 List<Number> 类型的对象，它实现了 Collection<Number> 接口，因此 nums 对象可以调用 Collection<Number> 接口的 addAll(Collection<? extends Number> c) 方法。第二，List<Integer> 是 List<? extends Integer> 的子类，而 List<? extends Integer> 是 Collection<? extends Number> 的子类，因此 List<Integer> 是 Collection<? extends Number> 的子类，根据子类替换原则，List<Integer> 可以作为 nums.addAll 的参数。



如何理解 List<Integer> 是 Collection<Number> 的子类，或者 List<Integer> 实现了 Collection<Number> 接口？请参考下图：



Java 的集合类框架包含了若干的接口及其实现类，下图可以帮助大家进一步了解集合类框架的层次结构和相互关系：



## 2.3 集合类中的泛型：PECS 原则

泛型通配符的用法的确有一点点复杂，Joshua Bloch 在著名的《Effective Java》中提出了 *PECS* 以帮助大家理解和记忆泛型通配符的用法，简洁有力。*PECS* 的意思是：Producer Extends, Consumer Super，即“读取时使用 `extends`，写入时使用 `super`”。下面我们分别讨论一下。



### 2.3.1 extends

我们以泛型 `List<? extends Number>` 为例，有如图2.5所示的类型层次关系。

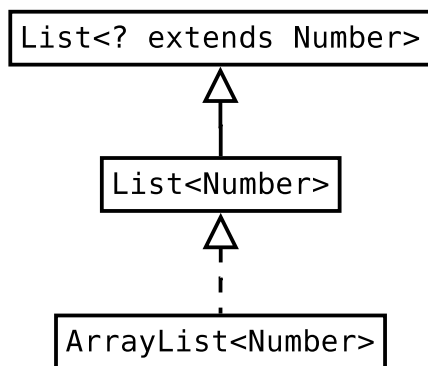


图 2.5: `List<? extends Number>` 的类层次关系

根据子类替换原则<sup>6</sup>，下面的语句都是合法的：

代码清单 2.14: `extends` 用法举例

```
1 List<? extends Number> nums = new ArrayList<Number>();
2 List<? extends Number> nums = new ArrayList<Integer>();
3 List<? extends Number> nums = new ArrayList<Double>();
```

我们从读取和写入两个方面考察 `nums`：

#### 2.3.1.1 读取

当调用 `nums.get(index)` 方法时，返回值的情形如下：

- 可以返回 `Number` 类型，因为 `nums` 中的数据元素是 `Number` 或者 `Number` 的子类（`Integer` 或者 `Double`），均可以安全的向上转型为 `Number` 类型。
- 可以返回 `Object` 类型，因为 `Object` 是 `Number` 的父类，向上转型是安全的。
- 不可以返回 `Integer`，因为 `nums` 作为一个引用，可能指向一个 `List<Double>`，如代码清单2.14的第 3 行所示。
- 不可以返回 `Double`，因为 `nums` 作为一个引用，可能指向一个 `List<Integer>`，如代码清单2.14的第 2 行所示。

因此有如下的一般推论：

**推论 2.1.** `List<? extends T>` 在读取时可以返回 `T` 或者 `T` 的父类对象，不可以返回 `T` 的子类对象。

<sup>6</sup>参见：<http://dz.sdut.edu.cn/blog/subaochen/2017/01/subtypes-substitution/>

### 2.3.1.2 写入

当调用 `nums.add()` 方法写入数据到 `List<? extends Number>` 时，情形如下：

- 不可以写入 `Number` 类型的对象。比如代码清单2.14的第2行，`nums` 指向一个 `List<Integer>` 对象，只能添加 `Integer` 或者 `Integer` 的子类对象，而 `Number` 是 `Integer` 的父类。
- 不可以写入 `Object` 类型的对象，因为 `nums` 可能指向一个 `List<Integer>` 类型的对象。
- 不可以写入 `Integer` 类型的对象，因为 `nums` 可能指向一个 `List<Double>` 类型的对象。
- 不可以写入 `Double` 类型的对象，因为 `nums` 可能指向一个 `List<Integer>` 类型的对象。

因此有如下的一般推论：

**推论 2.2.** `List<? extends T>` 不允许写入任何对象，即 `List<? extends T>` 不能用于写入数据的场合。

## 2.3.2 super

我们以泛型 `List<? super Number>` 为例，根据子类替换原则，以下的语句都是合法的：

代码清单 2.15: `super` 用法举例

```
1 List<? super Integer> nums = new ArrayList<Integer>();
2 List<? super Integer> nums = new ArrayList<Number>();
3 List<? super Integer> nums = new ArrayList<Object>();
```

### 2.3.2.1 读取

当调用 `nums.get(index)` 方法时：

- 无法保证结果是一个 `Integer`，因为 `nums` 可能指向一个 `List<Object>` 或者 `List<Number>`；
- 无法保证结果是一个 `Number`，因为 `nums` 可能指向一个 `List<Object>`；
- 结果可以是一个 `Object`，但是我们无法获知其具体的类型。

因此有如下的一般推论：

**推论 2.3.** `List<? super T>` 在读取时无法获知对象的具体类型，即 `List<? super T>` 不能用于读取数据的场合。

### 2.3.2.2 写入

调用 `nums.add` 方法写入时，情形如下：

- 可以写入 `Integer` 类型的对象，因为 `Integer` 是 `Integer`、`Number`、`Object` 的子类型。
- 同样的，可以写入 `Integer` 的子类对象。
- 不能写入 `Double` 类型的对象，因为 `nums` 可能指向一个 `List<Integer>`。
- 不能写入 `Number` 类型的对象，因为 `nums` 可能指向一个 `List<Integer>`。
- 不能写入 `Object` 类型的对象，因为 `nums` 可能指向一个 `List<Integer>`。

因此有如下的一般推论：

**推论 2.4.** `List<? super T>` 只允许写入 `T` 及 `T` 的子类对象。

### 2.3.3 PECS

根据以上的讨论，PECS 原则很自然就出来了：在读取时使用 *extends*，在写入是使用 *super*，即 *Producer Extends, Consumer Super*。

PECS 的一个著名例子是集合的复制，见代码清单2.16，注意到 `dest` 是目标集合用来接受数据，因此使用 `List<? super T>` 来定义，`src` 是源集合，用来读取数据，因此使用 `List<? extends T>` 定义。

代码清单 2.16: `Collections.copy` 方法

```
1 public class Collections {
2     public static <T> void copy(List<? super T> dest, List<? extends T> src)
3     {
4         for (int i=0; i<src.size(); i++)
5             dest.set(i,src.get(i));
6     }
7 }
```

**练习 2.2.** 请给出 `Collection.copy` 方法的几个具体应用实例 [ ?? [在第 ??页]]。

代码清单2.17可以考察你对 PECS 原则的理解程度，俗称烧脑测试。

代码清单 2.17: `PECTest.java`

```
1 package cn.edu.sdut.softlab;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
```

```

6  /**
7   * 本类综合演示了PECS原则.
8   * Created by subaochen on 17-1-8.
9   */
10 public class PECTest {
11     public static void main(String[] args) {
12         List<Number> listNumber_ListNumber = new ArrayList<Number>();
13         //List<Number> listNumber_ListInteger = new ArrayList<Integer>(); // 错误 -
            ArrayList<Integer>不是List<Number>的子类
14         //List<Number> listNumber_ListDouble = new ArrayList<Double>(); // 错误 -
            ArrayList<Double>不是List<Number>的子类
15         listNumber_ListNumber.add(3); // ok - 对于List<Number>, 可以添加整数数据
16
17         List<? extends Number> listExtendsNumber_ListNumber = listNumber_ListNumber; //
            ok - ArrayList<Number>是List<? extends Number>的子类型
18         List<? extends Number> listExtendsNumber_ListInteger = new ArrayList<Integer>();
19         List<? extends Number> listExtendsNumber_ListDouble = new ArrayList<Double>();
20
21         Number num1 = listExtendsNumber_ListNumber.get(0); // ok - List<? extends Number>
            的数据可以安全转型为Number
22         Object num2 = listExtendsNumber_ListNumber.get(0); // ok - List<? extends Number>
            的数据可以安全转型为Object
23         // Integer num3 = listExtendsNumber_ListNumber.get(0); // 错误 - List<? extends
            Number>的数据不可以安全转型为Integer
24         // Double num4 = listExtendsNumber_ListNumber.get(0); // 错误 - List<? extends
            Number>的数据不可以安全转型为Double
25
26
27         List<? super Number> listSuperNumber_ListNumber = listNumber_ListNumber;
28         //List<? super Number> listSuperNumber_ListInteger = new ArrayList<Integer>();
            // 错误 - Integer不是Number的父类型
29         //List<? super Number> listSuperNumber_ListDouble = new ArrayList<Double>(); //
            错误 - Double不是Number的父类型
30
31         // Number n1 = listSuperNumber_ListNumber.get(0); // 错误 -
            listSuperNumber_ListNumber可能指向List<Object>
32         // Integer n2 = listSuperNumber_ListNumber.get(0); // 错误 -
            listSuperNumber_ListNumber可能指向List<Object>
33         // Double n3 = listSuperNumber_ListNumber.get(0); // 错误 -
            listSuperNumber_ListNumber可能指向List<Object>
34         Object n4 = listSuperNumber_ListNumber.get(0); // ok - 但是, 我们不知道n4到底是什么
            类型的, 因此几无用处
35
36         //List<Integer> listInteger_ListNumber = new ArrayList<Number>(); // 错误 -
            ArrayList<Number>不是List<Integer>的子类
37         List<Integer> listInteger_ListInteger = new ArrayList<Integer>();
38         //List<Integer> listInteger_ListDouble = new ArrayList<Double>(); // 错误 -
            ArrayList<Double>不是List<Integer>的子类
39
40         //List<? extends Integer> listExtendsInteger_ListNumber = new ArrayList<Number>
            >(); // 错误 - Number不是Integer的子类型
41         List<? extends Integer> listExtendsInteger_ListInteger = new ArrayList<Integer>

```

```

    >());
42 //List<? extends Integer> listExtendsInteger_ListDouble = new ArrayList<Double>
    >()); // 错误 - Double不是Integer的子类型
43
44 List<? super Integer> listSuperInteger_ListNumber = new ArrayList<Number>();
45 List<? super Integer> listSuperInteger_ListInteger = new ArrayList<Integer>();
46 //List<? super Integer> listSuperInteger_ListDouble = new ArrayList<Double>();
    // 错误 - Double不是Integer的父类型
47
48
49 // 下面三行都存在编译错误, 原因是无法获知List<? extends Number>泛型变量真正指向的对象的数据
    类型。
50 //listExtendsNumber_ListNumber.add(3); // 错误 - 不能添加整数到可能的List<Double>,
    即使指向一个List<Number>也不行。
51 //listExtendsNumber_ListInteger.add(3); // 错误 - 不能添加整数到可能的List<Double>,
    即使指向一个List<Integer>也不行。
52 //listExtendsNumber_ListDouble.add(3); // 错误 - 不能添加整数到List<Double>
53
54 listSuperNumber_ListNumber.add(3); // ok - 可以添加整数到 List<Number> 或者 List<
    Object>
55
56 listInteger_ListInteger.add(3); // ok - 可以添加整数到未使用通配符的 List<Integer>
57
58 //listExtendsInteger_ListInteger.add(3); // 错误 - 不能添加整数到可能的List<X>, 无法
    获知X的数据类型
59
60 listSuperInteger_ListNumber.add(3); // ok - 允许添加整数到 List<Integer>, List<
    Number>, or List<Object>
61 listSuperInteger_ListInteger.add(3); // ok - 允许添加整数到 List<Integer>, List<
    Number>, or List<Object>
62 }
63 }

```

## 2.4 集合类的一般实用原则和注意事项

- 在代码中避免泛型类和原始类型的混用。比如 `List<String>` 和 `List` 不应该共同使用。这样会产生一些编译器警告和潜在的运行时异常。当需要利用 JDK 5 之前开发的遗留代码, 而不得不这么做时, 也尽可能的隔离相关的代码。
- 泛型类最好不要同数组一块使用。你只能创建 `new List<?>[10]` 这样的数组, 无法创建 `new List<String>[10]` 这样的。这限制了数组的使用能力, 而且会带来很多费解的问题。因此, 当需要类似数组的功能时候, 使用集合类即可。

## 第三章 lambda 表达式

### 3.1 为什么引入 lambda 表达式？

正式介绍 lambda 表达式之前，我们先看一个常见的情形，假设有类定义如代码清单3.1所示。

代码清单 3.1: Student.java

```
1 package cn.edu.sdut.softlab.lambda;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  *
8  * @author subaochen
9  */
10 public class Student {
11
12     private int age;
13     private String username;
14     private String email;
15
16     public static List<Student> testData;
17
18     static {
19         testData = new ArrayList<>();
20         testData.add(new Student("zhangsan", 18, "zhangsan@126.com"));
21         testData.add(new Student("lisi", 20, "lisi@163.com"));
22         testData.add(new Student("wangwu", 21, "wangwu@sohu.com"));
23         testData.add(new Student("zhaoliu", 23, "zhaoliu@126.com"));
24         testData.add(new Student("tianqi", 21, "tianqi@126.com"));
25         testData.add(new Student("wangba", 18, "wangba@126.com"));
26         testData.add(new Student("laojiu", 23, "laojiu@126.com"));
27     }
28
29     public Student(String username, int age, String email) {
30         this.age = age;
31         this.username = username;
```

```
32     this.email = email;
33 }
34
35 public int getAge() {
36     return age;
37 }
38
39 public void setAge(int age) {
40     this.age = age;
41 }
42
43 public String getUsername() {
44     return username;
45 }
46
47 public void setUsername(String username) {
48     this.username = username;
49 }
50
51 public String getEmail() {
52     return email;
53 }
54
55 public void setEmail(String email) {
56     this.email = email;
57 }
58
59 @Override
60 public String toString() {
61     return "Student{" + "age=" + age + ", username=" + username + ", email=" + email +
62         '}';
63 }
64
65 public void print() {
66     System.out.println(toString());
67 }
68
69 }
```

### 例 3.1. 列出大于指定年龄的学生<sup>1</sup>

**代码设计** 在 Client 中，我们可以设计一个 findStudentsOlderThan 方法如下：

```
1 public static void findStudentsOlderThan(int age) {
2     for (Student stu : testData) {
3         if (stu.getAge() > age) {
4             stu.print();
```

<sup>1</sup>本部分的完整示例请参见：<https://github.com/subaochen/java-tutorial-examples/tree/master/chap03/src/cn/edu/sdut/softlab/lambda>

```
5     }  
6     }  
7 }
```

**代码分析** 这段代码很容易理解。`testData` 是一个 `List<Student>` 集合,作为测试,我们在 `Student` 中初始化了 `testData`。`findStudentsOlderThan` 遍历所有的 `testData` 元素,找出大于指定年龄的学生并输出。

但是,这段代码也不够灵活,比如我们希望找出年龄在 20 和 25 之间的所有学生,就不得不重新改写上面的方法,或者重新编写一个新的方法:

```
1 public static void findStudentsAgeBetween(int minAge, int maxAge) {  
2     for (Student stu : testData) {  
3         if (stu.getAge() > minAge && stu.getAge() < maxAge) {  
4             stu.print();  
5         }  
6     }  
7 }
```

### 例 3.2. 隔离筛选条件

本质上,例3.1是根据指定的条件筛选学生。筛选方法是多样化的,有没有办法可以将筛选条件隔离出来呢?这是“设计模式”的重要原则:隔离变化。

**代码设计和分析** 我们可以设计一个独立的筛选接口: `CheckStudent` 如下:

```
1 public interface CheckStudent {  
2     /**  
3      * 检查学生是否满足筛选条件。  
4      *  
5      * @param stu 待检查的学生对象  
6      * @return 满足条件返回true, 否则返回false  
7      */  
8     boolean test(Student stu);  
9  
10 }
```

针对例3.1的情形,可以定义一个 `CheckStudent` 的实现类如下:

```
1 // 不好的实现  
2 public class StudentCheckerOlderThan implements CheckStudent {  
3  
4     @Override  
5     public boolean test(Student stu) {  
6         return stu.getAge() > 18;  
7     }  
8  
9 }
```



于是我们就可以在主类中这样实现对学生的条件筛选：

```
1 public static void main(String[] args) {
2     findStudents(new StudentCheckerOldThan());
3 }
4
5 public static void findStudents(CheckStudent checker) {
6     for(Student stu : testData) {
7         if(checker.test(stu)) {
8             stu.print();
9         }
10    }
11 }
```

可以看出,虽然我们在 `main` 中实现了对筛选条件的隔离,但是在 `StudentCheckerOldThan` 中却只能将筛选条件 (年龄大于 18) 写死了,很糟糕的设计! 如何避免写死筛选条件呢? 答案是在 `main` 方法中使用匿名类:

```
1 public static void main(String[] args) {
2     findStudents(new CheckStudent() {
3         public boolean test(Student stu) {
4             return stu.getAge() > 18;
5         }
6     });
7 }
```

这样我们多次调用 `findStudents` 方法时,可以传入不同的年龄得到不同的结果,也避免了定义一个写死了年龄的 `StudentCheckerOlderThan` 类。

Java8 进一步简化了匿名类的写法,这就是 `lambda` 表达式。

### 例 3.3. 使用 `lambda` 表达式简化匿名类的写法

如下所示:

```
1 public static void main(String[] args) {
2     findStudents(stu->stu.getAge() > 18);
3 }
```

可以看出,使用 `lambda` 表达式极大的简化了匿名类的写法。那么,如何理解这个 `lambda` 表达式呢?

实际上, `lambda` 表达式代表了一个匿名函数。每个 `lambda` 表达式分为三部分,如图3.1所示。

其中:

- 参数列表: 在函数体中使用到的参数列表,有几个就列出几个,使用逗号隔开多个参数,并使用小括号包围所有参数。如果只有一个参数,小括号可以省略。在列出参数时,也可以明确说明参数的数据类型,但是通常不需要这样做,Java 会根据

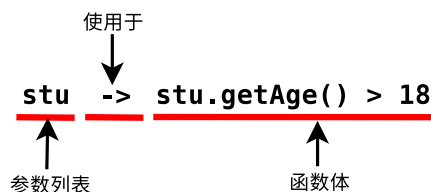


图 3.1: lambda 表达式的三个组成部分

上下文自动推断参数的类型。不明确说明参数类型的好处是，一个 lambda 表达式可以适用于多种类型的参数，有点泛型的味道了。如果参列表为空，使用 `()` 表示。

- “使用于”符号：“`->`”仅表示前面的参数列表在函数体中使用，没有更多的含义。
- 函数体：也就是 lambda 表达式的执行部分，根据传入的参数执行在函数体中给出的语句。如果语句不止一句，则需要使用大括号括起来。

我们知道函数的 4 个基本要素是：函数名、参数列表、返回值类型和函数体，lambda 表达式是一个匿名函数，参数列表和函数体都在 lambda 表达式中体现出来了，函数的返回值是如何决定的呢？在例 3.3 中，`findStudents` 方法的参数要求一个实现了 `CheckStudent` 接口类型的对象，因此 lambda 表达式作为一个匿名方法，其返回值是 `CheckStudent` 接口类型。实际上，Java 的 lambda 表达式的返回值要求是一个“接口函数”类型：首先，这是一个接口，其次，这个接口中有且只有一个未实现的方法（抽象方法），比如 `CheckStudent` 中的 `test` 方法。这样，lambda 表达式作为一个匿名方法，实际上是实现了这个函数接口，并在函数体中给出了函数接口中抽象方法的一个具体实现。

为了方便编写 lambda 表达式，Java8 新增了 `java.util.function` 包，其中包含了一些预定义的“接口函数”，可以“拿来”直接用，常见的接口函数如表 3.1 所示：

有了预定义的函数接口，例 3.2 中的 `CheckStudent` 接口就没有存在的必要了，`findStudents` 方法直接使用 `Predicate` 接口作为参数即可。

**例 3.4.** 使用 Java8 预定义的接口重新编写例 3.2。

**代码设计** 为了清晰起见，我们重新命名 `findStudents` 方法为 `findStudentsWithLambda`：

```
1 public static void findStudentsWithPredinedFunction(Predicate<Student> checker) {
2     for(Student stu : testData) {
3         if(checker.test(stu))
4             stu.print();
5     }
6 }
```

接口	接口函数	备注
@FunctionalInterface public interface Function<T,R>	R apply(T t)	T 为参数，R 为 lambda 表达式的返回值。
@FunctionalInterface public interface Consumer<T>	void accept(T t)	T 为参数，无需返回值。
@FunctionalInterface public interface Supplier<T>	T get()	没有参数，T 为返回值类型。
@FunctionalInterface public interface Predicate<T>	boolean test(T t)	T 为参数，返回一个 boolean 类型。

表 3.1: Java8 新增的函数接口

在 main 方法中，调用 findStudentsWithPredefinedFucntion 的方式和 find-Students 是一样的：

```
1 findStudentsWithPredinedFunction(stu -> stu.getAge() > 20);
```

**代码分析** 可见，借助于 Java 提供的预定义函数接口，我们在很多情况下就不需要自己再定义函数接口了。但是也需要注意到 Java8 预定义函数接口的不同应用场景：

- **Function**：往往用于将参数 T 转换为指定的返回类型 R。比如，我们只希望检查对象 T 的某个属性 R，则可以简单的借用 Fucntion 函数接口来实现。
- **Consumer**：只是对参数 T 执行某种操作（处理），比如打印输出，保存到某种介质（数据库）等。
- **Supplier**：当我们需要一个和输入条件无关的结论时，Supplier 很适合。
- **Predicate**：判断给定的参数对象 T 是否满足一定的条件。

**例 3.5.** 遍历学生集合，打印年龄大于 20 岁学生的 Email

**代码设计** 代码如下所示：

```
1 public static void findStudentsEmailByAge(
2     Predicate<Student> checker, // 筛选学生的条件
3     Function<Student, String> mapper, // 如何映射筛选出的学生对象
```

```
4     Consumer<String> action) { // 对于映射的部分如何处理
5     for(Student stu : testData) {
6         if(checker.test(stu)) {
7             String email = mapper.apply(stu);
8             action.accept(email);
9         }
10    }
```

调用方式:

```
1    findStudentsEmailByAge(
2        stu->stu.getAge() > 20,
3        stu -> stu.getEmail(),
4        email -> System.out.println(email));
```

**代码分析** 参见代码中的注释, 我们首先使用了 **Predicate** 函数接口来筛选结果集, 然后使用 **Function** 函数接口抽取 (映射) 我们感兴趣的部分, 最后使用 **Consume** 函数接口对我们感兴趣的部分进行处理。

上述的 **findStudentsEmailByAge** 方法还可以使用泛型进一步“通用化”处理, 如下:

```
1    public static <X> void findStudents(
2        Predicate<Student> checker,
3        Function<Student, X> mapper,
4        Consumer<X> action) {
5        for(Student stu : testData) {
6            if(checker.test(stu)) {
7                X info = mapper.apply(stu);
8                action.accept(info);
9            }
10        }
11    }
```

如此, 我们使用 **findStudents** 不仅可以处理 **Email** 地址, 还可以处理任意的 **Student** 中的属性了, 比如将 20 岁以上学生的姓名转换为大写字母:

```
1    findStudents(
2        stu -> stu.getAge() > 20,
3        stu -> stu.getUsername(),
4        username -> System.out.println(username.toUpperCase())
5    );
```

对于此种应用场景, 更好的解决方案是使用集合的流式处理模式, 参见 section §3.4。



编写 **lambda** 表达式时尤其要注意到, **lambda** 表达式要求返回的是一个函数接口类型, 请看下面的例子:



```
1 System.out.println(()->10); // 错误的用法，因为根据println的定义，Java无法推断出()->10
   实现了哪个函数接口
2 System.out.println((Supplier())->10); // 正确，明确了lambda表达式返回一个Supplier类型
   的函数接口
3 System.out.println(((Supplier())->10).get()); // 调用Supplier接口的get方法打印出
   lambda表达式的值：10
```

**练习 3.1.** 编写一个 lambda 表达式，计算两个整数的和、差、积。（参考答案参见解答 ?? [在第 ??页]）

## 3.2 lambda 表达式的类型

上面我们说 lambda 表达式是一个“匿名函数”，根据我们的经验，函数只有返回值、函数名、参数和函数体四个要素，那么 lambda 表达式的类型是怎么回事呢？实际上，我们可以把 lambda 表达式理解为“对象”，因为 lambda 表达式实现了“函数接口”中的唯一抽象方法，那么这个函数接口的类型就可以认为是 lambda 表达式的类型，Java8 称之为“目标类型”（target type）。

## 3.3 lambda 表达式的应用场景

如3.1所述，lambda 表达式可以用于替换匿名类，也就是说，几乎所有使用匿名类的地方都可以使用 lambda 表达式，前提条件是，匿名类实现的接口是一个函数接口。实际上，Java 中的匿名类接口很多是函数接口，比如线程中的 Runnable 接口，GUI 中的 ActionListener 接口等。

### 例 3.6. 使用 lambda 表达式替换线程中的匿名类

在引入 lambda 表达式之前，我们这样创建一个线程：

```
1 Thread myThread = new Thread( new Runnable () {
2     @Override
3     public void run() {
4         System.out.println("myThread created by an anonymous class.");
5     }
6 } );
```

借助于 lambda 表达式，创建线程就简单多了：

```
1 Thread myThread = new Thread(()->{
2     System.out.println("myThread created by lambda")
3 } );
```

注意第二个线程里的 `lambda` 表达式, 我们并不需要显式地把它转成一个 `Runnable`, 因为 `Java` 能根据上下文自动推断出来: 一个 `Thread` 的构造函数接受一个 `Runnable` 参数, 而传入的 `lambda` 表达式正好符合其 `run()` 函数, 所以 `Java` 编译器推断它为 `Runnable`。

从形式上看, `lambda` 表达式只是为你节省了几行代码。但将 `lambda` 表达式引入 `Java` 的动机并不仅仅为此。`Java8` 有一个短期目标和一个长期目标。短期目标是: 配合“集合类批处理操作”的内部迭代和并行处理(下面将要讲到); 长期目标是将 `Java` 向函数式编程语言这个方向引导(并不是要完全变成一门函数式编程语言, 只是让它有更多的函数式编程语言的特性), 也正是由于这个原因, `Oracle` 并没有简单地使用内部类去实现 `lambda` 表达式, 而是使用了一种更动态、更灵活、易于将来扩展和改变的策略(`invokedynamic`)<sup>2</sup>。



`lambda` 表达式实际上是把函数 A 作为函数 B 参数的艺术, 也就是说, 把函数 A 作为函数 B 的参数, 这样在函数 B 内部就可以使用函数 A 所提供的功能了。通常, 在函数 B 内部是一个循环或者迭代的过程, 而每一次循环或者迭代都会用到函数 A 进行相应的处理。

## 3.4 集合的流式处理

在 `Java8` 之前, 我们要遍历或者迭代集合类, 只能借助于 `while/for` 等循环机制, `Java8` 为每一个集合类增加了 `stream()` 方法, 将集合类的数据集转换为一个“流”(stream), 通过流的方式来操作集合类中的数据集。我们在??中已经看到了“流”的威力, 比如从键盘输入数字最终转换为一个浮点数的过程可以形象的表示为图 figure 3.2:

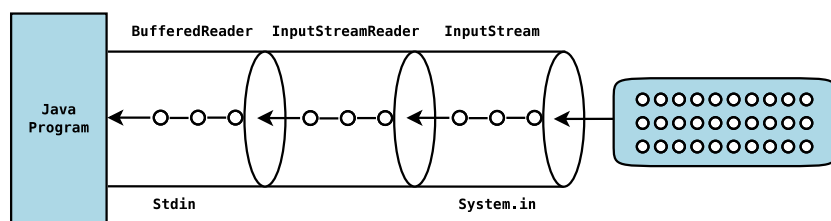


图 3.2: 从键盘输入的处理过程

比如从键盘输入了字符串“123.45”后, `InputStream`、`InputStreamReader`、`BufferedReader` 就像是三节管道一样首尾相接协同工作, 即 `InputStream` 的输出(二进制流)作为 `InputStreamReader` 的输入, `InputStreamReader` 的输出(字节流)作为 `BufferedReader` 的输入, `BufferedReader` 的输出是一个去掉了回车换行的

<sup>2</sup>参见: <http://blog.csdn.net/ioriogami/article/details/12782141>

字节流。由此可见，“流”是一个很适合数据的过滤、转换等处理的机制，而且非常灵活。Java8 为集合类引入“流”机制，极大的方便了集合类数据的处理。可以说，有了集合类的流式处理，基本上可以告别 for/while 循环了。

Java 为集合类的流提供了如表 table 3.2的基本方法。

方法	说明
<code>void forEach(Consumer&lt;? super T&gt; action)</code>	遍历集合数据，在每一个数据上执行 lambda 表达式 action
<code>&lt;R&gt; Stream&lt;R&gt; map(Function&lt;? super T,? extends R&gt; mapper)</code>	通过类型为 Function 的 lambda 表达式将输入类型 T 转换为类型 R，map 的输出是一个流
<code>Stream&lt;T&gt; filter(Predicate&lt;? super T&gt; predicate)</code>	通过类型为 Predicate 的 lambda 表达式过滤集合数据，filter 的输出是一个流
<code>&lt;R&gt; R collect(Supplier&lt;R&gt; supplier, BiConsumer&lt;R,? super T&gt; accumulator, BiConsumer&lt;R,R&gt; combiner)</code>	

表 3.2: Java8 的集合类 stream 方法

### 例 3.7. 通过 stream 方法遍历元素

forEach 接受一个 Consumer 类型的 lambda 表达式，即如何处理集合中的元素的“action”。

```
1 testData.stream().forEach((stu)->{System.out.println(stu)});
```

### 例 3.8. 通过 stream 方法过滤元素

filter 方法接受一个 Predicate 类型的 lambda 表达式，对集合中的每一个元素都执行 Predicate 函数接口中的 test 方法进行验证，其输出是一个经过过滤处理的流，因此可以作为 forEach 的输入：

```
1 testData.stream() // 包含所有数据的流
2     .filter(stu -> stu.getAge() > 20) // 包含大于20岁学生的流
3     .forEach(stu -> System.out.println(stu)); // 打印大于20岁的每一个学生
```

### 例 3.9. 通过 stream 方法转换元素

进一步的，filter 的输出流，可以通过 map 的数据转换处理后，在送给 forEach 处理。map 方法接受一个 Function 类型的 lambda 表达式来抽取我们感兴趣的数据，其结果是一个转换后的流：

```
1 // 使用stream筛选集合数据并抽取部分数据处理
2 testData.stream() // 包含所有数据的流
3     .filter(stu -> stu.getAge() > 18) // 包含大于20岁学生的流
4     .map(stu -> stu.getEmail())
5     .forEach(email -> System.out.println(email)); // 打印大于20岁学生的Email
```

### 例 3.10. 通过 stream 获得数据子集

在例 3.8 [在对页]时，我们是将过滤后的数据子集送到 `forEach` 中依次进行处理，也可以通过 `collect` 方法将数据子集“收集”起来重新放到一个集合中：

```
1 // 获得符合要求的数据子集
2 List<Student> list = testData.stream()
3     .filter(stu -> stu.getAge() > 18)
4     .collect(Collectors.toList());
5 System.out.println(list);
```

### 例 3.11. 通过 stream 获得数据子集的部分字段

进一步，我们也可以只“收集”感兴趣的部分：

```
1 // 获得符合要求的数据子集的部分字段收集起来
2 List<String> emails = testData.stream()
3     .filter(stu -> stu.getAge() > 20)
4     .map(stu -> stu.getEmail())
5     .distinct() // 去除重复内容
6     .collect(Collectors.toList());
7 System.out.println(emails);
8 // 也可以收集到Map中
9 Map<String,String> stuMap = testData.stream()
10     .filter(stu -> stu.getAge() > 20)
11     .distinct() // 去除重复内容
12     .collect(Collectors.toMap(stu -> stu.getUsername(), stu -> stu.getEmail()));
13 System.out.println(stuMap);
```

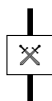
注意到 `Collectors.toMap` 方法的两个参数都是 `Function` 类型的 `lambda` 表达式，第一个 `lambda` 表达式生成了 `Map` 的 `key`，第二个 `lambda` 表达式生成了 `Map` 的 `value`。

### 例 3.12. 通过 stream 方法分组统计元素

比如我们统计大于 18 岁的各个年龄的人数：

```
1 // 统计指定数据子集
2 Map<Integer, Integer> r = testData.stream()
3     .filter(stu -> stu.getAge() > 18)
4     .map(stu -> new Integer(stu.getAge()))
5     .collect(Collectors.groupingBy(p -> p, Collectors.summingInt(p -> 1)));
6 System.out.println(r);
```





要注意到 `stream` 的哪些方法返回流: `filter`, `map`, `distinct` 等返回的都是一个新的流, 因此可以在这个新的流的基础上继续操作。

## 3.5 lambda 表达式的作用范围

本质上, `lambda` 表达式是一个匿名函数, 因此和匿名类有相似的作用范围: 在 `lambda` 表达式中可以获取“包围类”(即定义 `lambda` 表达式的类)中的变量。

### 例 3.13. lambda 表达式的作用范围

代码设计 参见代码清单3.2

代码清单 3.2: LambdaScopeTest.java

```
1 package cn.edu.sdut.softlab.lambda;
2
3 import java.util.function.Consumer;
4
5 /**
6  * 本类演示了Lambda的作用域。
7  *
8  * @author Su Baochen
9  */
10 public class LambdaScopeTest {
11
12     public int x = 0;
13
14     class FirstLevel {
15
16         public int x = 1;
17
18         void methodInFirstLevel(int x) {
19
20             // The following statement causes the compiler to generate
21             // the error "local variables referenced from a lambda expression
22             // must be final or effectively final" in statement A:
23             //
24             // x = 99;
25             Consumer<Integer> myConsumer = (y)
26                 -> {
27                 System.out.println("x = " + x); // Statement A
28                 System.out.println("y = " + y);
29                 System.out.println("this.x = " + this.x);
30                 System.out.println("LambdaScopeTest.this.x = "
31                     + LambdaScopeTest.this.x);
32             };
33
34         }
```

```
34     myConsumer.accept(x);
35
36 }
37 }
38
39 /**
40  * 程序执行入口。
41  *
42  * @param args 命令行参数
43  */
44 public static void main(String[] args) {
45     LambdaScopeTest st = new LambdaScopeTest();
46     LambdaScopeTest.FirstLevel fl = st.new FirstLevel();
47     fl.methodInFirstLevel(23);
48
49 }
50
51 }
```

**运行结果** 在 NetBeans IDE 中执行 LambdaScopeTest 结果如下：

```
run:
x = 23
y = 23
this.x = 1
LambdaScopeTest.this.x = 0
成功构建 (总时间: 0 秒)
```

**代码分析** 这个运行结果是显然的，不再赘述。

## 参考资料

- <http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
- <http://blog.csdn.net/ioriogami/article/details/12782141>

## 第四章 多线程 Java 程序设计

### 4.1 线程的基本概念

现代 CPU 都是多核心设计，应用程序如果不能充分利用 CPU 的多个核心就太浪费了。一般的，一个应用程序运行时表现为一个进程，这个进程占用一个 CPU 的核心。如果我们在这个进程中再发起几个所谓的“线程”去充分利用其他的 CPU 核心，这个应用程序就是一个“多线程”的，能够充分利用多核心 CPU 资源的的应用程序，如图4.1所示。比如我们常见的 **Web Server** 就是一个典型的多线程应用程序：每个请求都会衍生出多个线程分别获取文字、图片等等，然后再在主线程中合并所获取的 **HTML** 文档。

幸运的是，**Java** 是天生就支持多线程的程序设计语言，**Java** 中通过 **Thread** 类来描述一个线程。**Java** 应用程序启动时至少创建了一个线程，称为“主线程”，我们前面所接触的 **Java** 应用程序都是如此，大家可以在 **Idea** 的调试视图中可以看到 **Java** 应用程序创建的主线程。本章重点讨论如何编写多线程的 **Java** 应用程序。

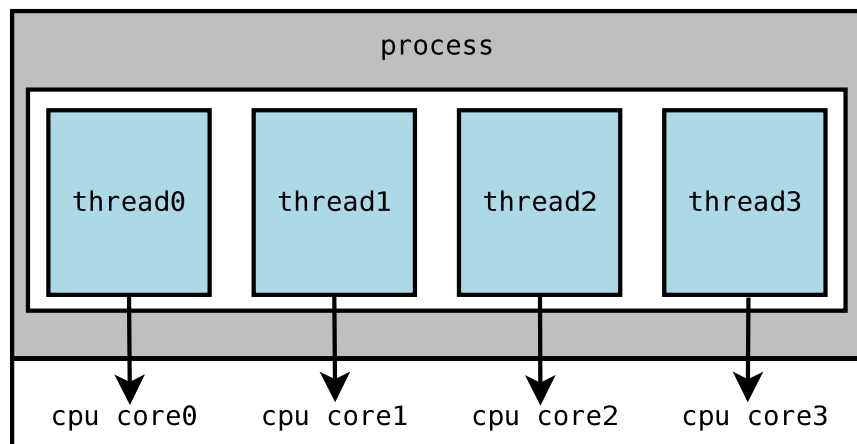


图 4.1: 多线程应用程序示例

## 4.2 线程的创建

Java 支持两种方式创建线程：

- 1. 实现 **Runnable** 接口，只要覆盖 Runnable 接口的唯一方法 run() 即可。
- 2. 继承 **Thread** 类，覆盖 Thread 类的 run() 方法。实际上，Thread 类实现了 Runnable 接口，因此从 Thread 继承下来自然也就实现了 Runnable 接口，这两种方法是殊途同归的。

通常建议通过 Runnable 接口来创建线程，因为 Java 是单继承的，如果从 Thread 继承下来，就无法继承其他类了，而实现 Runnable 接口就没有这个限制。但是无论哪种方式创建一个线程，都是通过 Thread 类提供的一些方法来管理线程，如表4.1所示。

方法	说明
void start()	告诉 Java 虚拟机可以启动指定的线程
static void sleep(long millis)	当前线程暂时“休眠”指定的毫秒数
void interrupt()	终止该线程
interrupted()	检测线程是否被中断？本方法执行后线程的中断标志被复位
isInterrupt()	检测线程是否被中断？本方法不操作中断标志符
void join()	暂停当前线程，等待 join 的线程执行完毕后才能继续执行
static Thread currentThread()	获得当前线程的引用
boolean isAlive()	线程是否处于活动状态
void setDaemon(Boolean on)	设置是否为后台线程
void setPriority(int newPriority)	设置线程优先级
void yield()	暂停当前线程（让出 cpu 时间），允许别的线程开始运行

表 4.1: Thread 的常见方法

例 4.1. 通过 Runnable 接口创建线程，参见代码清单4.1。

代码清单 4.1: RunnableThread.java

```
1 package cn.edu.sdut.softlab;
2
3 public class RunnableThread implements Runnable {
```

```
4
5 public static void main(String[] args) {
6     Thread thread = new Thread(new RunnableThread());
7     thread.start();
8 }
9
10 @Override
11 public void run() {
12     System.out.println("Thread is running...");
13 }
14 }
```

**例 4.2.** 通过继承 Thread 类创建线程，参见代码清单4.2。

代码清单 4.2: MyThread.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-2-10.
5  */
6 public class MyThread extends Thread {
7     @Override
8     public void run() {
9         System.out.println("MyThread is running...");
10    }
11
12    public static void main(String[] args) {
13        new MyThread().start();
14    }
15 }
```

**例 4.3.** 多线程示例

**代码设计** 参见代码清单4.3。

代码清单 4.3: MultiThreadDemo.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-2-10.
5  */
6 public class MultiThreadDemo implements Runnable {
7     @Override
8     public void run() {
9         System.out.println("Thread " + Thread.currentThread().getName() + " is running");
10        ;
11    }
12 }
```

```
12 public static void main(String[] args) {  
13     for(int i = 0; i < 10; i++) {  
14         Thread thread = new Thread(new MultiThreadDemo());  
15         thread.setName("Thread" + i); // 为了区分线程  
16         thread.start();  
17     }  
18 }  
19 }
```

**运行结果** 注意到，本示例程序每次执行的结果可能不一样，下面是其中的一个可能的执行结果：

```
Thread Thread1 is running  
Thread Thread0 is running  
Thread Thread2 is running  
Thread Thread3 is running  
Thread Thread5 is running  
Thread Thread4 is running  
Thread Thread6 is running  
Thread Thread7 is running  
Thread Thread8 is running  
Thread Thread9 is running
```

**代码说明** 本例中，我们通过一个 `for` 循环创建了 10 个线程并“依次”启动 (`start`)，但是仔细观察我们发现，程序的执行结果表明，这 10 个线程并不是依次顺序启动的！这是 Java 线程的重要特点：线程的启动和停止是 Java 虚拟机根据其调度算法来决定的，我们通过调用 `start()` 启动一个线程只是告诉 Java 虚拟机：现在可以启动这个线程了，至于线程真正的启动时间，取决于 Java 虚拟机的调度算法和当前 CPU 的运行状态。因此，`start/interrupt` 方法并不能立刻启动或者停止一个线程，仅仅是通知 Java 虚拟机可以启动或者停止一个线程而已。

### 4.2.1 使用 `sleep` 暂时中断线程的执行

`sleep` 是 `Thread` 类的一个静态方法，其用意是暂停当前线程的运行。在运用 `sleep` 的时候要注意两点：

1. `sleep` 的参数虽然精确到毫秒，但是 `sleep` 的暂停时间并非精确的毫秒数，我们也不能依赖 `sleep` 方法产生或者暂停精确的时间。

2. `sleep` 方法执行期间，如果有其他线程对当前线程执行了 `interrupt()` 方法，则会抛出 `InterruptedException` 异常<sup>1</sup>。

#### 例 4.4. 倒计时程序<sup>2</sup>

**代码设计** 设计一个倒计时 10s 的 Java 应用程序，参见代码清单4.4。

代码清单 4.4: Countdown.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-2-10.
5  */
6 public class Countdown {
7     public static final int COUNTER = 10;
8     public static void main(String[] args) throws InterruptedException {
9         for(int i = 0; i < COUNTER ; i++) {
10             System.out.println(COUNTER - i);
11             Thread.sleep(1000); // 暂停执行1000ms
12         }
13     }
14 }
```

**运行结果** 可以看出，每隔大约 1 秒屏幕打印出一个数字：

```
10
9
8
7
6
5
4
3
2
1
```

**代码分析** `Thread.sleep()` 方法暂停了当前线程，即 `Thread.sleep()` 所处的线程。本例中，虽然表面上我们没有创建任何线程，但是 Java 本身是天生支持多线程的，每个

<sup>1</sup>参见节 4.2.2 [在对页]。

<sup>2</sup>我们曾经在chapter 2使用 Queue 设计过一个类似的倒计时程序，参见<https://github.com/subaochen/java-tutorial-examples/blob/master/collections/src/cn/edu/sdut/softlab/CountDown.java>

Java 应用程序在执行时都会自动创建一个线程(即所谓的“主线程”),因此 `Thread.sleep()` 方法在这里暂停的就是主线程。

**练习 4.1.** 将例4.3中的线程使用 `sleep` 方法暂停一段时间(比如 1s),观察输出的变化。

### 4.2.2 使用 `interrupt` 终止线程的执行

启动一个线程是简单的,但是终止一个线程却没有想象中的那么容易。Java 不建议(未来可能会禁止)直接使用某个方法(比如 `stop()`)来终止线程<sup>3</sup>,而是建议通过“协商”机制来终止线程。也就是说,当我们需要终止线程时,调用 `interrupt()` 方法通知线程:嗨,你可以停下了!但是,被告知的线程是不是停下来,这要看线程是如何响应的:被告知的线程可以立刻退出,也可以处理掉一些手头的工作再停止,甚至可以完全不理睬!

具体的说,当我们调用一个线程的 `interrupt()` 方法时,有如下的两种情形 [?]:

1. 如果该线程处在可中断状态下,(比如调用了 `wait()`,或者 `Selector.select()`, `Thread.sleep()` 等会发生阻塞的 API,即可以抛出 `InterruptedException` 的方法),则设置现成的中断标志符为 `true`,并立即唤醒该线程,该线程同时会收到一个 `InterruptedException` 异常。此时如果该线程是阻塞在 io 操作上(比如读写文件、socket 等),则对应的资源会被关闭。如果该线程接下来不执行 `Thread.interrupted()` 方法去检测线程的中断状态(注意,不是 `interrupt()` 方法),那么该线程处理任何 io 资源的时候,都会发现这些资源已经关闭。解决的办法就是调用一下 `interrupted()`,不过这里需要程序员自行根据代码的逻辑来设定,根据自己的需求确认是否可以直接忽略该中断,还是应该马上退出。
2. 如果该线程处在不可中断状态下(即线程正在运行的代码不会抛出 `InterruptedException`),那么 Java 只是设置一下该线程的 `interrupt` 状态标志为 `true`,其他事情都不会发生,如果该线程之后会调用阻塞 API(抛出 `InterruptedException` 异常的方法),那到时候线程会马上跳出,并抛出 `InterruptedException`,接下来的事情就跟第一种状况一致了。如果不会调用阻塞 API,那么这个线程就会一直执行下去,就像 `interrupt()` 方法没有执行一样。除非你就是要实现这样的线程,一般高性能的代码中肯定会有 `wait()`, `yield()` 之类出让 cpu 的函数,不会发生后者的情况。

总而言之,在调用了线程的 `interrupt()` 方法后,如何终止线程是由线程自己决定的!每个线程需要根据自己的业务逻辑来决定如何处理 `interrupt()` 调用,这里需要权衡性能

---

<sup>3</sup>具体原因参见: <http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>



和响应时间的矛盾，在提升软件的执行效率和提升软件的人机响应速度上达到一个较好的平衡。

通常，下面的情形需要调用线程的 `interrupt()` 方法：

- 点击了“停止”按钮，如果存在的话；
- 点击了“退出”或者“关闭”按钮，应该对所有线程发出 `interrupt()` 调用；

如何设计一个可以优雅退出或者终止的线程呢？通常有如下的两种策略：

1. 在线程的业务逻辑中检查线程的中断状态：

```
1 Thread thread = new Thread("my thread") {
2     @Override
3     public void run() {
4         while(true) {
5             doSomething(); // 线程的业务逻辑
6             if(Thread.interrupted()){
7                 doClear(); // 清理工作
8                 break;
9             }
10        }
11    }
12 };
13 thread.start();
```

2. 在线程的业务逻辑中捕获 `InterruptedException`：

```
1 private void doSomething throws InterruptedException {
2     // 业务逻辑处理
3     ....
4     if(Thread.interrupted()) { // 检查线程的中断状态。每个在线程中调用到的方法建议如此处理
5         throw new InterruptedException();
6     }
7 }
8
9 Thread thread = new Thread("my thread") {
10     @Override
11     public void run() {
12         while(true) {
13             try {
14                 doSomething(); // 线程的业务逻辑方法，在这个方法中检查线程的状态并抛出
15                             InterruptedException
16             } catch (InterruptedException e) {
17                 doClear(); // 清理工作
18                 break;
19             }
20         }
21     }
22 }
```

```
19     }
20 }
21 };
22 thread.start();
```

通常建议采取第二种策略更合适一些：可以通过异常处理机制集中处理中断请求。采取第二种策略要求在设计线程中可能调用的方法时要注意在适当的时候检查当前线程的状态，否则异常处理就形同虚设了。

比如例4.4中，我们忽略了 `InterruptedException`，只是简单的把这个异常再抛给了 Java 虚拟机，这显然不是一个“负责”的程序应该有的态度。其实在这种情况下，只需要简单的 `return` 即可，参见代码清单4.5。

代码清单 4.5: Countdown2.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-2-10.
5  */
6 public class Countdown2 {
7     public static final int COUNTER = 10;
8     public static void main(String[] args){
9         for(int i = 0; i < COUNTER ; i++) {
10             System.out.println(COUNTER - i);
11             try {
12                 Thread.sleep(1000); // 暂停执行1000ms
13             } catch (InterruptedException e) {
14                 return; // 简单return即可，不然呢？
15             }
16         }
17     }
18 }
```

**例 4.5.** 使用 `interrupt()` 终止线程示例<sup>4</sup>

**代码设计** 参见代码清单4.6。

代码清单 4.6: ThreadInterruptDemo.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-2-12.
5  */
6 public class ThreadInterruptDemo implements Runnable {
7     public static void main(String[] args) throws InterruptedException {
```

<sup>4</sup>本示例借鉴自：[https://www.tutorialspoint.com/javaexamples/thread\\_interrupt.htm](https://www.tutorialspoint.com/javaexamples/thread_interrupt.htm)

```
8      Thread thread = new Thread(new ThreadInterruptDemo(), "My Thread");
9      thread.start();
10     Thread.sleep(5000); // ❶
11     System.out.println("in main - interrupt other thread");
12     thread.interrupt(); // ❷
13     System.out.println("in main - leaving");
14 }
15
16 @Override
17 public void run() {
18     while (true) {
19         try {
20             printDot();
21         } catch (InterruptedException e) {
22             System.out.println("in run() - " + Thread.currentThread().getName() + " is
                interrupted");
23             break;
24         }
25     }
26
27     System.out.println("in run() - thread done");
28 }
29
30 private void printDot() throws InterruptedException { // ❸
31     System.out.print(".");
32     Thread.sleep(1000);
33     if (Thread.interrupted()) throw new InterruptedException();
34 }
35 } //
```

❶ 主线程休眠 5000ms

❷ 终止子线程 My Thread

❸ 在 run() 方法中调用的方法，最好检测线程终止事件并抛出 InterruptedException

**运行结果** 运行本程序，会每隔 1s 打印一个句点，打印 5 个句点后程序结束：

```
.....in main - interrupt other thread
in main - leaving
in run() - My Thread is interrupted
in run() - thread done
```

**代码说明** 本例中，主线程启动 My Thread (My Thread 为线程的名字) 线程后，休眠了 5000ms，以便 My Thread 线程打印出几个句点。大约 5000ms 后，主线程通知 My Thread 终止运行。My Thread 线程立刻被从休眠中唤醒并抛出了 InterruptedException，于是程序流程转向了 catch 代码块，打印出“in run() - My Thread is interrupted”后终止了无限循环。

从本例很明显可以看出，`interrupt()` 方法并不会立刻终止线程，线程如何终止以及何时终止，这是线程自己的事情。

### 4.2.3 使用匿名内部类创建线程

有的时候，我们需要创建一个临时线程去执行特定的任务，这个时候可以使用匿名内部类简化代码，参见代码清单4.7。

代码清单 4.7: AnonThreadDemo.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 17-2-12.
5  */
6 public class AnonThreadDemo {
7     public static void main(String[] args) {
8         new Thread(new Runnable(){
9
10             @Override
11             public void run() {
12                 System.out.println(Thread.currentThread().getName() + " is running");
13             }
14         }, "MyThread").start();
15     }
16 }
```

## 4.3 线程间的数据共享和协作

在多线程编程中，如何存取**共享且可变**的数据是个基本的问题。由于多个线程的执行顺序无法预测，且可能存在交叉执行的情形，对于共享的可变数据基本都是采取“串行”方式处理，即通过互斥和同步实现了线程间的数据共享和协作。互斥提供了对竞争资源的保护，同步则实现了多线程的协同工作。

在这里要注意两点：

1. 共享 (shared)：只有在线程间共享的对象才需要特别处理，线程内部的私有对象不会被其他线程修改，也就不会造成破坏性的影响了。
2. 可变 (mutable)：如果共享的对象不可变（即所谓的 `immutable`），那么也不需要特别处理。

由于不可变 (immutable) 的对象不需要进行同步和互斥的处理, 因此在多线程编程实践中, 应该尽量将类设计为不可变的, 下面是一些常见的设计不可变类的方法和技巧<sup>a</sup>:

✕

- 不提供 `setter` 方法, 保护对象的私有属性不会被修改。
- 使用 `private` 和 `final` 限制属性。
- 使用 `final` 限制类被继承, 防止子类覆盖方法。进一步的, 将构造方法设置为 `private` 的, 只允许通过工厂类创建对象。
- 如果属性引用了可变的对象, 使用如下的办法保护可变对象:
  - 不提供可能修改可变对象的方法。
  - 不直接分享可变对象; 如果要分享, 创建可变对象的 `copy` 来分享, 这样万一其他线程改变了可变对象的 `copy` 的状态, 也不会影响原始的可变对象。

<sup>a</sup>详情参见: <https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>

比如电脑的打印机是一个竞争资源, 当一个应用程序正在打印文档时, 其他应用程序只能排队等待打印机空闲后才能继续打印。对竞争资源的保护可以通过“加锁”的方式实现, 当一个线程在使用竞争资源时, 对竞争资源加锁, 其他线程只有在该线程解锁后才能继续使用竞争资源。我们在现实中也是通过加锁的方式实现对竞争资源的保护的, 比如火车上的卫生间是一个竞争资源, 每个旅客可以看做是一个线程, 当你进入卫生间一定会锁门的吧? 加锁的目的是只有当前线程完整的使用完竞争资源, 其他线程才能获得继续使用竞争资源的机会。

下面我们讨论 Java 多线程编程中最常见的三种同步情形。

### 4.3.1 使用 `synchronized` 保护竞争资源

当同时有多个线程使用同一个对象 (竞争资源) 时, 如果不加特别处理, 竞争资源的状态可能会遭到破坏而导致最终结果不正确, 这是因为多个线程的执行顺序可能会有交错, 而且很难事先预测线程的执行顺序。

我们首先看一个简单的计数器的例子, 参见代码清单4.8。为了模拟多线程可能交错运行的情况, 我们故意将一句简单的 `c++` 扩展为三条语句, 并在语句之间使用多个 `sleep()` 方法暂停线程的执行。同时, 为了观察多线程交错运行的情况, 在每一条语句之后都打印出了相关信息 (为了清晰起见, 在 `increment()` 中的输出以 `#` 开头, 在 `decrement()` 中的输出以 `*` 开头)。在代码清单4.9中, 我们启动了两个线程。两个线程共享一个对象 `counter`, 其属性 `c` 初值为 0, 其中 `t1` 线程对 `c` 执行加 1 操作, `t2`

线程对 `c` 执行减 1 操作，因此理论上两个线程结束后，最终的 `c` 值应该为 0。但是我们多次运行 `CounterClient` 会发现结果可能很难预测，下面是其中一次执行的情况：

```
#increment()
*decrement()
#increment() - after get value from c to temp:0
*decrement() - after get value from c to temp:0
*decrement() - after temp--:-1
*decrement() - after store c to new value:-1
#increment() - after temp++:1
#increment() - after store c to new value:1
finally, c is 1
```

从执行结果已经可以看出，`t1` 和 `t2` 两个线程的语句是交错执行的。我们逐句分析一下执行结果：

1. `#increment() - after get value from c to temp:0`，此时执行的是 `t1` 线程的 `temp = c` 操作，即将 `counter` 对象的 `c` 的值赋值为 `temp`，此时 `c` 和 `temp` 的值为 0；
2. `*decrement() - after get value from c to temp:0`，同上，此时执行的是 `t1` 线程的 `temp = c` 操作，即将 `counter` 对象的 `c` 的值赋值为 `temp`，由于此时 `c` 为 0，因此 `temp` 的值为 0；注意到 `temp` 是一个局部变量；
3. `*decrement() - after temp--:-1`，执行 `t2` 线程的 `temp--` 语句，此时 `temp` 的值是 -1；
4. `*decrement() - after store c to new value:-1`，执行 `t2` 线程的 `c = temp` 语句，此时 `counter` 对象的 `c` 值为 -1；到此为止，`t1` 线程执行完毕所有语句。
5. `#increment() - after temp++:1`，执行 `t1` 线程的 `temp++` 语句，此时 `temp` 的值是 1；
6. `#increment() - after store c to new value:1`，执行 `t1` 线程的 `c = temp` 语句，保存 `counter.c` 的值为 1，这也是最终的 `counter.c` 的值。问题出现了，由于 `t1` 线程在执行 `increment()` 方法时被中断，恰好在中断期间 `increment` 方法中引用的属性或者对象（这里是 `counter.c`）被其他线程（这里是 `t2`）修改了，而 `t1` 线程重新被执行时，`t1` 线程并不知道发生了这种情况，因此只能继续使用旧的过时数据（`c = 0`，其实此时 `c` 已经被 `t2` 修改为 -1 了），造成最终结果不正确。

另外一次执行的情况可能是这样的：

```
#increment()
*decrement()
#increment() - after get value from c to temp:0
*decrement() - after get value from c to temp:0
*decrement() - after temp--:-1
#increment() - after temp++:1
#increment() - after store c to new value:1
*decrement() - after store c to new value:-1
finally, c is -1
```

也有可能是正确的结果：

```
#increment()
*decrement()
#increment() - after get value from c to temp:0
#increment() - after temp++:1
#increment() - after store c to new value:1
*decrement() - after get value from c to temp:1
*decrement() - after temp--:0
*decrement() - after store c to new value:0
finally, c is 0
```

请读者自行逐句分析每一次执行的结果。

#### 代码清单 4.8: Counter.java

```
1 package cn.edu.sdut.softlab.contention;
2
3 import java.util.Random;
4
5 /**
6  * Created by subaochen on 17-2-13.
7  */
8 public class Counter {
9     private int c = 0;
10
11     public synchronized void increment() {
12         System.out.println("#increment()");
13         sleep();
14         int temp = c;
15         System.out.println("#increment() - after get value from c to temp:" + c);
16         sleep();
17         temp++;
18         System.out.println("#increment() - after temp++:" + temp);
```

```
19     sleep();
20     c = temp;
21     sleep();
22     System.out.println("#increment() - after store c to new value:" + temp);
23 }
24
25 public synchronized void decrement() {
26     System.out.println("*decrement()");
27     sleep();
28     int temp = c;
29     System.out.println("*decrement() - after get value from c to temp:" + temp);
30     sleep();
31     temp--;
32     System.out.println("*decrement() - after temp--:" + temp);
33     sleep();
34     c = temp;
35     sleep();
36     System.out.println("*decrement() - after store c to new value:" + temp);
37 }
38
39 public int value() {
40     return c;
41 }
42
43 private void sleep() {
44     Random random = new Random();
45     try {
46         Thread.sleep(random.nextInt(1000));
47     } catch (InterruptedException e) {
48         return;
49     }
50 }
51 }
```

代码清单 4.9: CounterClient.java

```
1 package cn.edu.sdut.softlab.contention;
2
3 import java.util.Random;
4
5 /**
6  * Created by subaochen on 17-2-13.
7  */
8 public class CounterClient {
9
10     public static void main(String[] args) {
11         final Counter counter = new Counter();
12         Thread t1 = new Thread(new Runnable() {
13             @Override
14             public void run() {
15                 counter.increment();
```



```
16     }
17     });
18
19     Thread t2 = new Thread(new Runnable() {
20         @Override
21         public void run() {
22             counter.decrement();
23         }
24     });
25
26     t1.start();
27     t2.start();
28     try {
29         t1.join(); // main线程要等待t1执行完毕才能继续执行
30         t2.join(); // main线程要等待t2执行完毕才能继续执行
31     } catch (InterruptedException e) {
32         // ignore
33     }
34
35     System.out.println("finally, c is " + counter.value());
36 }
37 }
```

我们看到，问题的根源在于，在 `increment` 或者 `decrement` 方法执行过程中，对于共享变量 `counter.c` 必须保证在整个方法执行过程中状态保持一致，或者说，在方法执行过程中，必须保证竞争变量 `counter.c` 变量不会被其他线程修改，即通过“互斥锁”对竞争资源进行串行化处理。解决这个问题有两种思路：

1. 将 `increment` 方法和 `decrement` 方法变为原子方法，即 `increment` 方法和 `decrement` 方法必须一次执行完毕，不允许被打断，具体的方法是在 `increment()` 方法和 `decrment()` 方法前使用 `synchronized` 关键词修饰。当使用 **`synchronized`** 关键词修饰一个方法时，该方法将成为原子方法，只有该方法执行完毕后才出让 `cpu` 时间。修改后的 `Couter.java` 类参见代码清单??，测试类没有变化<sup>5</sup>，不再赘述。执行测试类的结果为：

---

<sup>5</sup>但是要注意到，在我们提供的示例程序包中实现了同步的测试类在 `cn.edu.sdut.softlab.contention.syncned1` 包中，不要和包 `cn.edu.sdut.softlab.contention` 中的 `CounterClient.java` 相混淆：虽然两个类的内容完全一样，但是引用的 `Counter` 类不同。

```
#increment()
#increment() - after get value from c to temp:0
#increment() - after temp++:1
#increment() - after store c to new value:1
*decrement()
*decrement() - after get value from c to temp:1
*decrement() - after temp--:0
*decrement() - after store c to new value:0
finally, c is 0
```

可以看出，**increment** 方法和 **decrement** 方法是顺序依次执行的，这就是 **synchronized** 修饰方法时达到的效果。实际上，使用 **synchronized** 修饰方法，实际上是给该方法的对象加了一把锁，只有该方法执行完毕释放锁之后，其他线程才有机会获得该对象其他被 **synchronized** 方法的执行权<sup>6</sup>。

2. 给变量 `counter.c` 加锁，即 `counter.c` 在被某方法调用时，首先将 `counter.c` 保护起来，在该方法执行完之前（实际上是释放锁定之前），其他线程不允许修改 `counter.c` 的值，参见代码清单??。

代码清单 4.10: 加锁属性的 Counter.java

```
1 package cn.edu.sdut.softlab.contention.synced2;
2
3 import java.util.Random;
4
5 /**
6  * Created by subaochen on 17-2-13.
7  */
8 public class Counter {
9     private Integer c = 0;
10
11     public void increment() {
12         sleep();
13         synchronized (c) {
14             sleep();
15             c++;
16         }
17     }
18 }
```

<sup>6</sup>打一个比方，一个对象（object）就像一个大房子，里面有很多房间（方法）。其中有的房间是上锁的（synchronized 方法），有的房间没有上锁（普通方法）。上锁的房间需要通过钥匙打开进入，钥匙只有一把，挂在房子的大门口，**这把钥匙可以打开所有上锁的房间**。我们把执行方法的线程比作进入房间的人，那么如果一个人要进入一间上锁的房间，必须从门口拿到钥匙才行。一旦此人拿到了钥匙，别的人就没有机会进入大房子中其他上锁的房间了：记住，**钥匙只有一把**！只有此人从上锁的房间离开，把钥匙重新挂在大门口，别人才有机会拿到钥匙进入上锁的房间。即便是此人要再次进入上锁的房间，也要先把钥匙归还重新去拿。简单的说，钥匙的使用原则是“只有一把，随用随借，用完即还”。当然，其他人进入没有上锁的房间是没有限制的。

```
17     sleep();
18 }
19
20 public void decrement() {
21     sleep();
22     synchronized (c) {
23         sleep();
24         c--;
25     }
26     sleep();
27 }
28
29 public int value() {
30     return c;
31 }
32
33 private void sleep() {
34     Random random = new Random();
35     try {
36         Thread.sleep(random.nextInt(1000));
37     } catch (InterruptedException e) {
38         return;
39     }
40 }
41 }
```

使用 `synchronized` 的修饰方法时，要注意被修饰的方法应该尽量短小精悍，尤其是执行频率高的方法，更应该注意到这一点，否则可能严重影响应用程序的执行效率。本例中在 `increment` 方法和 `decrement` 方法中使用 `sleep()` 方法故意制造延迟，在实际应用中是不可取的，请读者注意。

使用 `synchronized` 修饰方法时，被修饰的方法执行完毕会自动为其中的竞争资源建立所谓的“happens-before” [?, P330，先行发生原则]<sup>a</sup>关系，即保证本方法对竞争资源的修改对所有其他线程是可见的。

<sup>a</sup>happens-before 在虚拟机设计中是一个很重要的概念，详情参见：Java Language Specification#17.4.5

### 4.3.2 使用 `ThreadLocal` 隔离竞争资源

有时，我们希望对于某个对象，每个线程拥有一份独立的拷贝而不受其他线程干扰，这在容器的设计中非常常见。比如 `Session` 的实现、`Context`（上下文）的实现等等。也就是说，线程之间共享一个数据结构（便于主线程有机会管理各个线程），但是又相互隔离，即线程能够互不影响的操作此数据结构。最容易想到的解决方案是使用 `Map`，其中的 `key` 是线程 ID 或者名称，`value` 是所需要的数据结构，这样线程在使用 `Map` 的时候，首先根据各自的 ID 或者名称查到此数据结构就可以进行操作了。Java 提供了类

似的 `ThreadLocal` 机制帮助我们简化这一工作。

我们先看一个不使用 `ThreadLocal` 的例子，参见：代码清单4.11。

代码清单 4.11: `UnsafeThreadDemo.java`

```
1 package cn.edu.sdut.softlab.threadlocal;
2
3 /**
4  * Created by subaochen on 17-2-15.
5  */
6 public class UnsafeThreadDemo {
7     private static int sum = 0;
8
9     public static void main(String[] args){
10         for (int i = 0; i < 10; i++) {
11             TestThread thread = new TestThread(String.valueOf(i));
12             thread.start();
13         }
14     }
15
16     static class TestThread extends Thread {
17
18         private String name;
19
20         public TestThread(String name) {
21             super(name);
22             this.name = name;
23         }
24
25         @Override
26         public void run() {
27             sleep(50);
28             for (int i = 0; i < 10; i++) {
29                 sum += 5;
30             }
31             System.out.println("unsafe thread " + name + ":" + sum);
32         }
33
34         private void sleep(int ms) {
35             try {
36                 Thread.sleep((long) (Math.random() * ms));
37             } catch (InterruptedException ex) {
38                 ex.printStackTrace();
39             }
40         }
41     }
42 }
```

在这里，`sum` 是主线程的一个变量，此变量在创建的 10 个线程中都会用到，我们看一下输出结果：

```
unsafe thread 6:50
unsafe thread 8:100
unsafe thread 4:245
unsafe thread 1:245
unsafe thread 2:245
unsafe thread 3:295
unsafe thread 7:345
unsafe thread 5:395
unsafe thread 0:445
unsafe thread 9:495
```

可以看出,由于主线程的 `sum` 变量是共享的,测试线程中的 `sum` 是相互影响的。如何解决这个问题呢?除了本节开头介绍的使用 `Map` 的思路,我们可以借助于 `ThreadLocal` 来实现,参见代码清单4.12。

代码清单 4.12: SafeThreadLocalDemo.java

```
1 package cn.edu.sdut.softlab.threadlocal;
2
3 /**
4  * Created by subaochen on 17-2-15.
5  */
6 public class SafeThreadLocalDemo {
7     static ThreadLocal<Integer> sum = new ThreadLocal<Integer>() {
8         @Override
9         protected Integer initialValue() {
10             return new Integer(0);
11         }
12     };
13
14     public static void main(String[] args) {
15         for (int i = 0; i < 10; i++) {
16             TestThread thread = new TestThread(String.valueOf(i));
17             thread.start();
18         }
19     }
20
21     static class TestThread extends Thread {
22
23         String name;
24
25         public TestThread(String name) {
26             super(name);
27             this.name = name;
28         }
29     }
30 }
```

```
30     @Override
31     public void run() {
32         for (int i = 0; i < 10; i++) {
33             sleep(50);
34             sum.set(sum.get() + 5);
35         }
36         System.out.println("safe thread " + name + ":" + sum.get());
37     }
38
39     private void sleep(int ms) {
40         try {
41             Thread.sleep((long) (Math.random() * ms));
42         } catch (InterruptedException ex) {
43             ex.printStackTrace();
44         }
45     }
46 }
47 }
```

执行结果如下：

```
safe thread 8:50
safe thread 5:50
safe thread 6:50
safe thread 3:50
safe thread 9:50
safe thread 0:50
safe thread 4:50
safe thread 7:50
safe thread 1:50
safe thread 2:50
```

因此，虽然 `sum` 还是主线程的共享变量，但是子线程（测试线程）就像独立拥有 `sum` 一样，不再相互影响，这也就是 `ThreadLocal` 的本意：线程本地变量，即通过 `ThreadLocal` 保护的变量，就像是子线程的本地变量一样。



`ThreadLocal` 通常用于保护线程的全局共享对象，即当其他线程使用此共享对象时，就像独立拥有此共享对象一样。容器技术中大量使用 `ThreadLocal` 来提供场景（上下文，`Context`）保护和场景切换。要注意的是，`ThreadLocal` 的作用并不是同步（`synchronized`），同步的作用是其他线程分时共享一个对象，这和 `ThreadLocal` 所要达到的效果正好相反。

### 4.3.3 使用 wait、notify 实现多线程的同步和协调

多线程的同步和协调其实是指线程间通信技术,Java 在 Object 根类中提供了 wait()、notify()、notifyAll () 方法,再配合 section 4.3.1 中的相关技术,可以很好的实现线程间的通信,参见表4.2。

方法名	说明
public final void wait() throws InterruptedException	线程暂停,直到调用 wait 方法的对象执行了 notify() 或者 notifyAll() 才被重新唤醒
public final void notify()	唤醒因为调用该对象的 wait() 方法处于暂停状态的一个线程。如果多个线程调用了该对象的 wait() 方法,则根据 JVM 的具体实现选择其一唤醒
public final void notifyAll()	唤醒所有因为调用该对象的 wait() 方法处于暂停状态的线程,但是所有线程需要竞争对该对象的访问权

表 4.2: Object 中关于线程间通信的方法

下面通过经典的“生产者/消费者”问题看一下 Java 解决线程间通信的基本方法。

#### 例 4.6. 多线程间通信示例

**代码设计** 设计一个仅能存储一个整数的 Buffer,参见代码清单4.13,设计一个线程使用生产者类 (Producer),参见代码清单4.15,用于往 Buffer 中存入(生产)数据。设计一个线程使用消费者类 (Consumer),参见代码清单4.14,用于从 Buffer 读取(消费)数据。约束条件:生产者在 Buffer 放入数据后,要通知消费者及时拿走;消费者从 Buffer 读取数据后,要及时通知生产者继续制造数据。

代码清单 4.13: Buffer.java

```

1 package cn.edu.sdut.softlab.sync;
2
3 /**
4  * Created by subaochen on 17-2-13.
5  */
6 public class Buffer {
7     private int value;
8     private boolean isEmpty = true;
9
10    public int get() {
11        return value;
12    }
13

```

```
14 public void set(int value) {
15     this.value = value;
16 }
17
18 public boolean isEmpty() {
19     return isEmpty;
20 }
21
22 public void setEmpty(boolean empty) {
23     isEmpty = empty;
24 }
25 }
```

代码清单 4.14: Consumer.java

```
1 package cn.edu.sdut.softlab.sync;
2
3 import java.util.Random;
4
5 /**
6  * Created by subaochen on 17-2-13.
7  */
8 public class Consumer implements Runnable {
9     private Buffer buffer;
10    public Consumer(Buffer buffer) {
11        this.buffer = buffer;
12    }
13
14    @Override
15    public void run() {
16        Random random = new Random();
17        while(true) {
18            synchronized (buffer) {
19                while (buffer.isEmpty()) {
20                    try {
21                        buffer.wait();
22                    } catch (InterruptedException e) {
23                        return;
24                    }
25                }
26
27                System.out.println("consumer value: " + buffer.get());
28                buffer.setEmpty(true);
29                buffer.notifyAll();
30            }
31        }
32    }
33 }
```

代码清单 4.15: Producer.java



```
1 package cn.edu.sdut.softlab.sync;
2
3 /**
4  * Created by subaochen on 17-2-13.
5  */
6 public class Producer implements Runnable {
7     private Buffer buffer;
8     private int counter = 0;
9
10    public Producer(Buffer buffer) {
11        this.buffer = buffer;
12    }
13
14    @Override
15    public void run() {
16        while(counter < 5) {
17            synchronized (buffer) {
18                while (!buffer.isEmpty()) {
19                    try {
20                        buffer.wait(); // not producer.wait!
21                    } catch (InterruptedException e) {
22                        return;
23                    }
24                }
25
26                buffer.set(counter++);
27                buffer.setEmpty(false);
28                System.out.println("producer value: " + buffer.get());
29                buffer.notifyAll();
30            }
31        }
32    }
33 }
```

代码清单 4.16: Client.java

```
1 package cn.edu.sdut.softlab.sync;
2
3 /**
4  * Created by subaochen on 17-2-13.
5  */
6 public class Client {
7     public static void main(String[] args) throws InterruptedException {
8         Buffer buffer = new Buffer();
9         Producer p = new Producer(buffer);
10        Thread pt = new Thread(p);
11        pt.start();
12
13        Consumer c = new Consumer(buffer);
14        Thread ct = new Thread(c);
15        ct.start();
16    }
17 }
```

```
16
17     Thread.sleep(5000); // 让两个线程最多运行5s
18     pt.interrupt();
19     ct.interrupt();
20 }
21 }
```

**运行结果** 测试代码如代码清单4.16所示，运行 Client 的结果如下：

```
producer value: 0
consumer value: 0
producer value: 1
consumer value: 1
producer value: 2
consumer value: 2
producer value: 3
consumer value: 3
producer value: 4
consumer value: 4
```

**代码说明** 从运行结果可以看出，两个线程的协调工作很成功，producer 和 consumer 总是成对出现，达到了预期的目的。

在使用 Object 的 wait/notify 机制实现线程间通讯的时候，要特别注意几点：

- 使用 synchronized 保护竞争资源，这里是 buffer 对象。因此无论是在 Producer 还是 Consumer 线程中，几乎总是以：synchronized(obj) {...} 开头。
- 使用 while 循环检测 wait 条件，只要不满足设定的条件则继续等待。
- 当满足事务逻辑的时候，要设置 wait 条件，并通知其他相关线程。

简单的说，下面的伪代码可以看作使用 wait/notify 机制的模板：

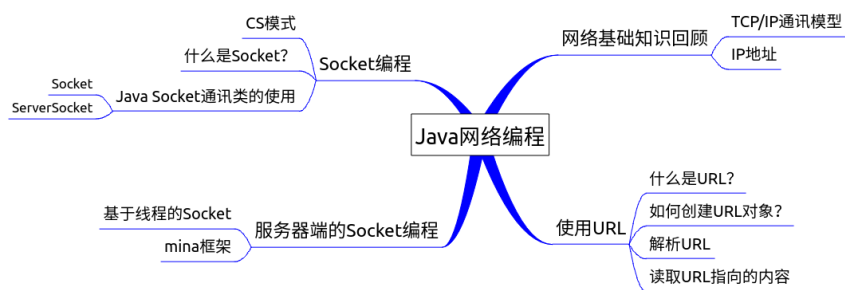
```
1 synchronized (obj) {
2     while (obj.waitCondition())
3         obj.wait();
4
5     // 如果无需等待，则处理事物逻辑，并通知其他线程
6     ...
7     obj.setWaitCondition();
8     obj.notifyAll();
9 }
```



正确使用 `wait/notify` 比较困难，也需要非常小心。但是从 JDK 1.5 开始，Java 提供了更高级的并发工具：`Executor Framework`、`Concurrent Collection` 和 `Synchronizer`，它们可以完成以前必须借助于 `wait/notify` 才能实现的各项工作，因此建议将 `wait/notify` 作为理解线程间通信的良好示范，但是实际应用时尽量使用 JDK 1.5 中提供的并发工具，避免使用 `wait/notify` [?, 第 69 条，并发工具优先于 `wait` 和 `notify`]。

**练习 4.2.** 将例4.6中的 `Buffer` 替换为一个 `Queue`，重新实现整个代码逻辑。

# 第五章 网络编程



## 5.1 网络通讯基础知识

在讲述 Java 的网络编程之前，我们迅速的回顾一下网络通讯的基础知识，读者如果已经熟悉此部分内容可跳过。

现代的互联网是基于 TCP/IP 的四层通讯模型的,如图5.1所示。虽然我们使用 java.net

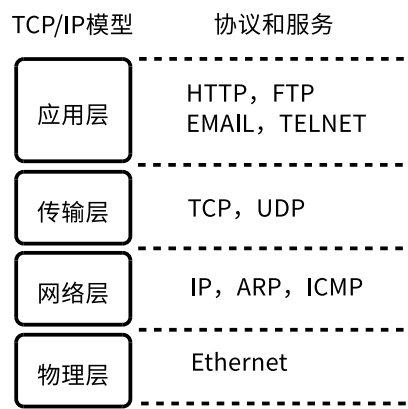


图 5.1: TCP/IP 四层通讯模型

包编写的 Java 网络应用程序位于应用层，但是了解 TCP、IP 的基本概念有助于我们选择合适的类包编写应用程序。

### 5.1.1 TCP

TCP(Transmission Control Protocol) 是一种可靠的传输控制协议。TCP 协议通过建立可靠的网络链接保证数据在传输过程中不会丢失，顺序也不会乱掉，因此当我们需要可靠的网络数据传输时就要选择 TCP 协议，比如负责网页传输的 HTTP 协议，负责文件传输的 FTP 协议，负责邮件传输的 Email 协议，负责远程登录的 Telnet/ssh 协议等：你一定不希望收到的是破损的文件或者邮件的，对吧？

但是，TCP 协议建立可靠的网络链接也是要付出代价的，即相对于“不太可靠”的 UDP 而言，TCP 的传输效率要低一些。

### 5.1.2 UDP

UDP (User Datagram Protocol) 是一种“不太可靠”的传输控制协议，即 UDP 并不保证数据发出后，接收端一定能够收到，也不保证数据传输的顺序。这种“不太可靠”的数据传输方式在并不要求百分百数据准确性的场合大有用武之地，比如网络视频，即使数据在传输中丢掉了一点点数据，并不影响视频的流畅和质量。

由于 UDP 协议不需要维护可靠的传输通道，UDP 协议的传输效率要比 TCP 高一些。

### 5.1.3 IP 地址

网络上的每一台设备都有一个唯一的地址编号，即 IP 地址，网络中的路由器就是根据数据包中的目的 IP 地址和源 IP 地址构建了数据传输的路由（通道）。常见的 IP 地址是 32 位的，每 8 位使用句点隔开，比如 210.44.176.123。

### 5.1.4 端口 (port)

我们知道，现代操作系统都是多任务操作系统，可以同时运行多个应用程序，每一个应用程序可以看做这台计算机提供的一种服务。如果这些应用程序提供网络服务，比如 HTTP，FTP 服务，那么如何区分在一台计算机上面运行着的多个网络应用程序呢？单纯依赖 IP 地址是不够的，需要借助以端口（Port）的概念，如图5.2所示。当数据到达计算机后，计算机会根据应用程序的不同将数据分发到不同端口，这就要求通信双方事先约定好使用的端口号。端口号是一个 16 位的数字，其范围为 0~55635，但是 0~1024 号端口通常被称作“熟知端口”，即这些端口很多已经用于大家约定俗成的网络通讯服务，比如 80 端口用于 HTTP 协议传输，21 用于 FTP 等等，因此我们自己编写的 Java 网络应用程序尽量避免使用 1024 以下的端口号，以免和熟知端口冲突。

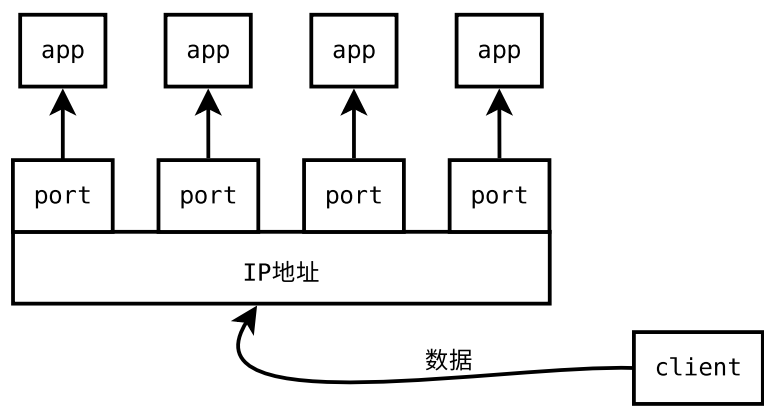


图 5.2: 端口示意图

## 5.2 使用 URL

### 5.2.1 什么是 URL ？

URL (Uniform Resource Locator) 是网络资源的唯一地址，即通常所说的“网址”。我们可以根据 URL 查找网络资源（文字、图片、视频等），`java.net.URL` 类表示了一个 URL。一个 URL 包含包含了如图5.3所示的几部分：

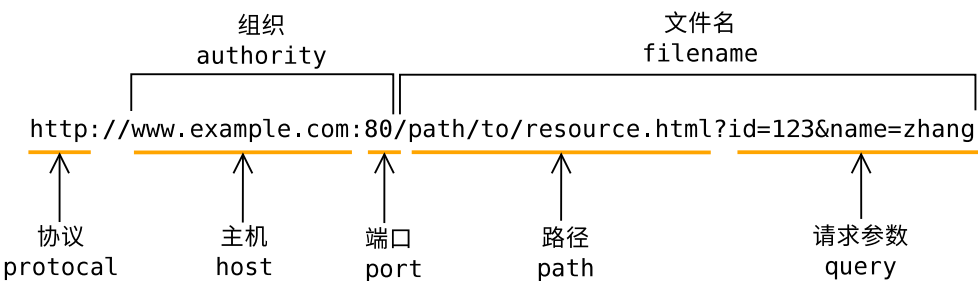


图 5.3: URL 示意图

- 协议 (protocol)：使用何种协议请求这个网络资源？`http` 用于请求超文本文件，只是常见的网络协议之一，此外还有 `ftp`、`email`、`telnet` 等各种网络协议。协议和主机部分通过`://` 隔开。
- 主机(host)：网络资源位于哪台网络设备，即主机中？主机往往由主机名和域名联合组成，如图5.3所示，`www` 是一台提供 WEB 服务的主机的主机名，`example.com` 是这台主机所在的域名，`www.example.com` 是 URL 的主机部分。
- 端口：(port)：如 section 5.1.4所述，端口决定了主机将 URL 请求转发到哪个应用程序来处理。对于常见的“熟知端口”，比如 `http` 协议的 `80` 端口，通常在 URL 中可以略去。

- 文件名 (**file**): 所请求的具体资源 (文件、图片等) 在主机的存储路径 (含文件名)。这里的文件名是一个广义的概念, 其一, 文件名包括了路径和文件的具体名称; 其二, **URL** 中的文件名也可能仅仅是一个目录, 但是要知道, **WEB** 服务器会自动根据配置情况附加一个 **index.html** 或者类似的具体文件名。
- 请求参数 (**query**): 在交互式网络应用程序中, 往往需要附带额外的参数来进一步确定所请求的网络资源, 通过“?” 隔开文件名和请求参数 (**Query String**)。

此外需要注意到, 在 **Java** 的 **URL** 类中, 还给出了 **authority** 的概念, 即主机和端口合称为 **authority**。

为什么称作 **authority** 呢? 其实一个完整的 **URL** 是这样子的<sup>a</sup>:

`scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]`

也就是说, 在一个 **URL** 中还包含 **user:password** 部分, 因此 **authority** 应该包含三部分:

1. 可选的用户名和密码
2. 必须的主机
3. 可选的端口号

也就是说, 这三个部分决定了用户是否有权限访问所给定的主机 (和具体的资源没有关系): 使用什么用户名和密码可以访问主机的哪个端口, 因此这三个部分联合起来又叫做 “**authority**”, 即权威认证。

<sup>a</sup>[https://en.wikipedia.org/wiki/Uniform\\_Resource\\_Identifier](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier)

## 5.2.2 创建 URL

**java.net.URL** 类的构造方法提供了常见的创建 **URL** 对象的构造方法, 如表5.1所示。

### 5.2.2.1 创建绝对 URL

创建 **URL** 对象的最简单方式, 是使用我们常见的 **URL** (网址) 的形式作为参数, 比如<sup>1</sup>:

```
1 URL url = new URL("http://www.example.com/doc/tutorial/networking.html?id=101");
```

<sup>1</sup>完整示例参见:<https://github.com/subaochen/java-tutorial-examples/tree/master/network/src/cn/edu/sdut/softlab>

方法名	说明
<code>public URL(String spec) throws MalformedURLException</code>	根据给定的 URL 字符串创建 URL 对象
<code>public URL(URL context, String spec) throws MalformedURLException</code>	根据给定的上下文和字符串创建 URL 对象
<code>public URL(String protocol, String host, int port, String file) throws MalformedURLException</code>	根据跟定的协议、主机、端口号和文件名创建 URL 对象
<code>public URL(String protocol, String host, String file) throws MalformedURLException</code>	根据跟定的协议、主机和文件名创建 URL 对象

表 5.1: URL 的构造方法



注意到参数：“http://www.example.com/doc/tutorial/networking.html?id=101”是一个完整的 URL 字符串，叫做“绝对 URL”，类似于我们描述文件名的时候，称带完整路径信息的文件名为“绝对路径”。

### 5.2.2.2 创建相对 URL

所谓的相对 URL，是指在给定的 URL 的基础上再附加所缺少的 URL 元素创建一个 URL 对象，类似于“相对路径”的概念，比如：

```
1 URL url = new URL("http://www.example.com"); // 创建一个绝对URL
2 URL tutUrl = new URL(url, "doc/tutorial/networking.html?id=101"); // 创建一个相对URL
```

在创建相对 URL 的时候，如果第一个参数为 null，则构造方法会将第二个参数看做“绝对 URL”来处理；如果第二个参数为绝对 URL，则会忽略第一个参数，比如：

```
1 URL url = new URL(null, "http://www.example.com/doc"); // 创建一个绝对URL对象
2 URL url = new URL(url, "http://www.example.com"); // 创建一个绝对URL对象
```

**练习 5.1.** 使用绝对 URL 和相对 URL 两种方式，分别创建 https://github.com/subaochen/java-tutorial-examples/tree/master/network/src/cn/edu/sdut/softlab 的 URL 对象。

### 5.2.2.3 使用协议、主机等创建 URL

有时我们获得的是 URL 的各个组成部分：协议、主机、文件名等，可以根据这些组成部分方便的创建 URL 对象，比如：

```
1 URL url = new URL("http", "www.example.com", "doc/tutorial/networking.html?id=101");
```

## 5.2.3 解析 URL

URL 类提供了一系列的 getXXX 方法帮助我们解析给定的 URL 对象的各个部分，如代码清单5.1所示，请读者参照图5.3并根据此示例程序的运行结果仔细体会 URL 的各个组成部分是如何划分的。

代码清单 5.1: ParseURL.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.net.URL;
4
```

```
5 /**
6  * Created by subaochen on 17-2-16.
7  */
8 public class ParseURL {
9     public static void main(String[] args) throws Exception {
10
11         URL aURL = new URL("http://example.com:80/docs/books/tutorial"
12             + "/index.html?name=networking#DOWNLOADING");
13
14         System.out.println("protocol = " + aURL.getProtocol());
15         System.out.println("authority = " + aURL.getAuthority());
16         System.out.println("host = " + aURL.getHost());
17         System.out.println("port = " + aURL.getPort());
18         System.out.println("path = " + aURL.getPath());
19         System.out.println("query = " + aURL.getQuery());
20         System.out.println("filename = " + aURL.getFile());
21         System.out.println("ref = " + aURL.getRef());
22     }
23 }
```

运行结果如下：

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING
```

### 5.2.4 读取 URL

创建 URL 对象后，我们可以通过 URL 类的 **openStream()** 方法获得一个到指定 URL 的 **InputStream** 流，然后就可以和操作本地文件一样方便的读取 URL 所指向的内容了，如代码清单5.2所示。

代码清单 5.2: URLReader.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.io.BufferedReader;
4 import java.io.InputStreamReader;
5 import java.net.URL;
6
7 /**
```

```
8 * Created by subaochen on 17-2-16.
9 */
10 public class URLReader {
11     public static void main(String[] args) throws Exception {
12
13         URL url = new URL("https://raw.githubusercontent.com/subaochen/java-tutorial-
14             examples/master/network/sample/hello.txt");
15         BufferedReader in = new BufferedReader(
16             new InputStreamReader(url.openStream()));
17
18         String inputLine;
19         while ((inputLine = in.readLine()) != null)
20             System.out.println(inputLine);
21         in.close();
22     }
23 }
```

运行结果如下<sup>2</sup>:

hello, java!



读取 URL 的另外方式是使用 `URLConnection` 类提供的方法, 同时 `URLConnection` 还提供了更丰富的交互功能, 详情参见:

<https://docs.oracle.com/javase/tutorial/networking/urls/connecting.html>

## 5.3 Socket 编程

网络通讯的基本形式是“端对端”通信, 即所谓的“客户端-服务器模式”(Client Server: CS), 如图5.4所示。在端对端通信中, 客户端和服务端首先建立通讯链接,

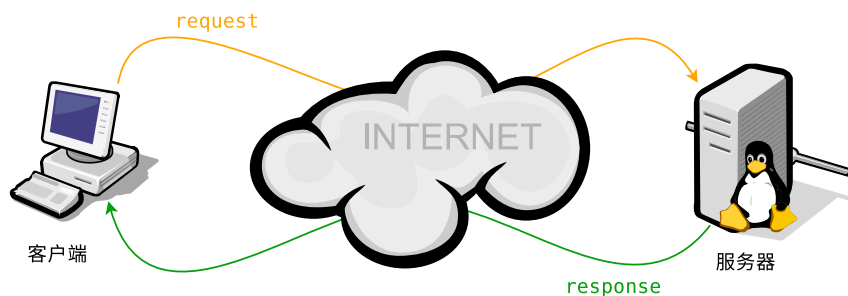


图 5.4: 客户-服务器通信模型

然后客户端发送请求 (request) 到服务器端, 服务器端返回响应 (response) 到客户

<sup>2</sup>是的, 不要怀疑, 获取的内容就这么简单! 读者可自行访问<https://raw.githubusercontent.com/subaochen/java-tutorial-examples/master/network/sample/hello.txt>验证一下。

端，实现了一次数据交换。这很像我们打电话的过程：首先拨号呼叫对方，如果对方摘机应答（或者手机上按下应答键），则双方通话的通道就建立起来了，然后就可以你一言我一语的通过电话交谈了。任何一方挂机，则整个通话过程（通信过程）就结束了。

### 5.3.1 什么是 Socket？

在端对端的通信中，通信的双方不仅需要知道对方的主机 IP 地址，还需要知道对方应用使用的端口号，这是建立端对端通信的必要基础，因此将 **IP 地址和端口号合称为 Socket**（插口），如图5.5所示。基于 Socket 概念的编程通常被称为“Socket 编

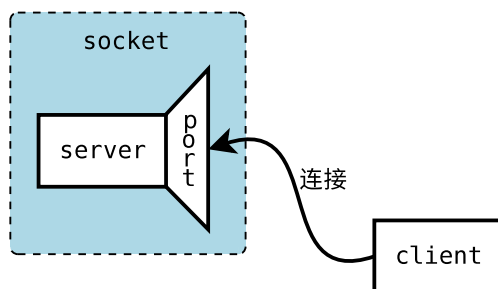


图 5.5: Socket 示意图

程”，java.net 包中提供了 Socket、ServerSocket 等类封装了 Socket 的基本操作，可以方便的实现客户端和服务器的 Socket 编程。

### 5.3.2 使用 Socket

下面我们通过一个实现了 Echo 协议的例子来说明 Java Socket 编程的基本思路。如代码清单5.3所示的 EchoServer.java 是服务器端，如代码清单5.4所示的 EchoClient.java 是客户端。客户端每次向服务器端发送字符串时，服务器端仅仅是把收到的字符串原样返回到客户端，即所谓的 echo 操作。

本示例的运行测试方法：首先启动 EchoServer，再启动 EchoClient。注意到 EchoServer 需要一个命令行参数<sup>3</sup>：端口号，这里假设为 2000；EchoClient 需要两个命令行参数，主机 IP 地址和服务器端端口号，在这里主机使用 localhost，服务器端端口号为 2000。本示例程序的服务器端 EchoServer 是静默运行的，即启动后没有给出任何提示。客户端 EchoClient 运行后可以在终端窗口输入内容，一个可能的测试结果如图5.6所示。

Java Socket 的基本原理如图5.7所示，服务器端（这里是 EchoServer）创建 ServerSocket 对象后，通过 serverSocket 对象的 accept 方法等待客户端发起连接请求。也就是说，SocketServer 是一个被动的服务器端，当 SocketServer 启动后

<sup>3</sup>设置命令行参数的方法参见：?? [在第 ??页]

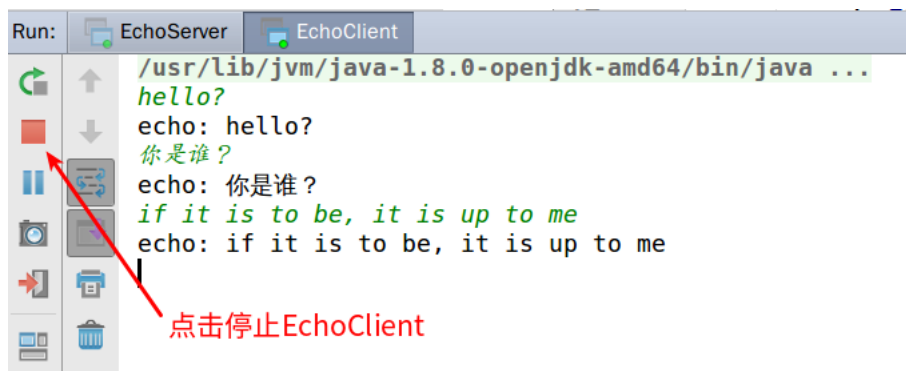


图 5.6: EchoClient 的运行结果

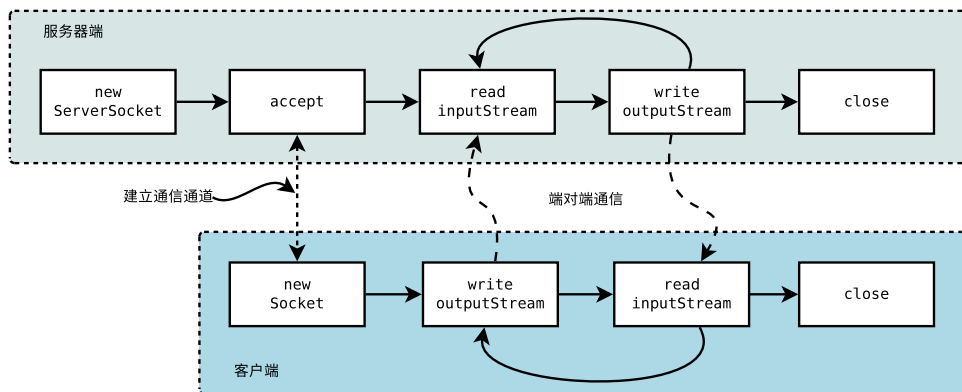


图 5.7: Java Socket 通讯示意图

会阻塞在 `accept` 方法，一直在等待客户端发起连接请求。一旦有客户端连接成功，则创建一个描述此端对端通信的 `Socket` 对象（这里是 `clientSocket`），通过此 `socket` 对象即可获得一个从客户端读取数据的 `InputStream`（这里是 `in`），以及可以写入客户端的 `OutputStream`（这里是 `out`）。注意到 `serverSocket`、`clientSocket`、`in`、`out` 都是需要用完即关闭的资源，因此在本例中我们使用 `try-with-resources`<sup>4</sup>结构帮助我们自动关闭打开的资源。

客户端 `EchoClient` 的过程类似，首先创建一个 `Socket` 对象（这里是 `echoSocket`）。注意到我们使用服务器端的 IP 地址和端口号创建的 `echoSocket`，因此 `echoSocket` 会试图根据我们提供的参数发起到服务器端的连接请求，连接成功则建立了客户端和服务端通信的通道，这就是 `echoSocket`。通过 `echoSocket` 我们可以获得从服务器端读取数据的 `InputStream`（这里是 `in`），往服务器端写入数据的 `OutputStream`（这里是 `out`），然后不断将从键盘输入的字符串送往服务器端即可。

一般的 Linux 发布版中都提供了 `Socket` 测试工具，包括服务器端和客户端。在 `ubuntu` 中可以简单的执行命令：

```
$ sudo apt-get install socket
```

安装 `socket` 测试工具。安装完成后，启动 `EchoServer`，可以在另外一个终端如下方式测试 `EchoServer` 的功能：

```
$ socket -v localhost 2000
inet: connected to localhost port 2000 (cisco-sccp)
hello?
hello?
你是谁?
你是谁?
我们交个朋友吧!
我们交个朋友吧!
```

`socket` 工具作为编写服务器端的测试工具很方便，当然，`socket` 也可以作为服务器端运行来测试客户端应用程序，读者可以自行探索。

另外一个简单的客户端工具是 `telnet`，比如：

<sup>4</sup>如果不熟悉 `try-with-resources` 结构，请参照 ?? [在第 ??页]



```
$ telnet localhost 2000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello
if it is to be, it is up to me
if it is to be, it is up to me
```

代码清单 5.3: EchoServer.java

```
1 package cn.edu.sdut.softlab;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9 import java.util.Scanner;
10
11 /**
12  * Created by subaochen on 17-2-17.
13  */
14 public class EchoServer {
15     public static void main(String[] args) throws IOException {
16
17         if (args.length != 1) {
18             System.err.println("Usage: java EchoServer <port number>");
19             System.exit(1);
20         }
21
22         int portNumber = Integer.parseInt(args[0]);
23
24         try (
25             ServerSocket serverSocket = new ServerSocket(portNumber); //❶
26             Socket clientSocket = serverSocket.accept(); //❷
27             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
28             //❸
29             Scanner in = new Scanner(clientSocket.getInputStream()); //❹
30         ) {
31             String inputLine;
32             while ((inputLine = in.nextLine()) != null) { //❺
33                 System.out.println("from client:" + inputLine);
34                 out.println(inputLine); //❻
35             }
36         } catch (IOException e) {
```

```

36         System.out.println("Exception caught when trying to listen on port "
37             + portNumber + " or listening for a connection");
38         System.out.println(e.getMessage());
39     }
40 }
41 } //

```

❶ 创建 `ServerSocket` 对象

❷ 等待客户端发起连接请求

❸ 获得写入客户端的 `outputStream`

❹ 获得从客户端读取数据的 `inputStream`

❺ 从客户端读取一行，如果存在的话

❻ 将从客户端读取的一行字符串发送回客户端

#### 代码清单 5.4: EchoClient.java

```

1  package cn.edu.sdut.softlab;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.io.PrintWriter;
7  import java.net.Socket;
8  import java.net.UnknownHostException;
9  import java.util.Scanner;
10
11  /**
12   * Created by subaochen on 17-2-17.
13   */
14  public class EchoClient {
15      public static void main(String[] args) throws IOException {
16
17          if (args.length != 2) {
18              System.err.println(
19                  "Usage: java EchoClient <host name> <port number>");
20              System.exit(1);
21          }
22
23          String hostName = args[0];
24          int portNumber = Integer.parseInt(args[1]);
25
26          try (
27              Socket echoSocket = new Socket(hostName, portNumber); // ❶
28              PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true); //
29              BufferedReader in =
30                  new BufferedReader(
31                      new InputStreamReader(echoSocket.getInputStream())); // ❷
32              Scanner stdIn = new Scanner(System.in); // ❸
33          ) {

```



```

34     String userInput;
35     while ((userInput = stdIn.nextLine()) != null) {
36         out.println(userInput); // ④
37         System.out.println("echo: " + in.readLine()); // ⑤
38     }
39     } catch (UnknownHostException e) {
40         System.err.println("Don't know about host " + hostName);
41         System.exit(1);
42     } catch (IOException e) {
43         System.err.println("Couldn't get I/O for the connection to " + hostName);
44         System.exit(1);
45     }
46 }
47 }//

```

① 根据 socket server 的主机 IP 地址和端口号创建一个到 socket server 的 socket 对象

② 获得写入到服务器端的 OutputStream

③ 获得从服务器读取数据的 InputStream

④ 标准输入

⑤ 写入到服务器端

⑥ 读取服务器端的响应并显示在屏幕上

**练习 5.2.** 使用 Socket 编写一个服务器端和客户端，客户端给出一个算术表达式，服务器端返回计算结果<sup>5</sup>。

### 5.3.3 \* 编写健壮的服务端 Socket 应用

在编写服务器端的 Socket 应用程序时要考虑以下的问题：

- 如何响应多个客户端的连接请求？通常的做法是针对每个客户发起一个独立的线程。
- 如何灵活的处理不同的业务逻辑，即客户端和服务端端的通讯协议？这是服务器端 Socket 编程的重要内容，不好的架构往往不恰当的提高了软件的复杂度，降低了软件的可维护性。
- 网络 I/O 和消息的编码、解码处理。
- 错误处理。网络通讯环境是复杂和易出错的，保证服务器端 SocketServer 总是能够正确响应不是一件很容易的事情。

有很多第三方组织开发了一些帮助开发者处理以上问题的框架，其中比较著名的是 Apache<sup>6</sup>组织的 mina<sup>7</sup>。Apache mina 2（2 指 mina 的第二个大的版本号，是目前最新版本）

<sup>5</sup>算术表达式求值可以参考：?? [在第 ??页]

<sup>6</sup><http://www.apache.org>

<sup>7</sup><http://mina.apache.org>

是一个开发高性能和高可伸缩性网络应用程序的网络应用框架。它提供了一个抽象的事件驱动的异步 API，可以使用 TCP/IP、UDP/IP、串口和虚拟机内部的管道等传输方式。Apache mina 2 可以作为开发网络应用程序的一个良好基础。下面就 mina 的开发做一简单介绍<sup>8</sup>。

### 5.3.3.1 建立 mina 开发环境

**下载 mina** 首先需要下载 mina 的合适版本，我们采用最新的版本即可，本书写作时 mina 的最新版本是 2.0.16，可以从这里下载：<http://mina.apache.org/mina-project/downloads.html>。下载后（这里是下载到 ~/downloads 目录下）解压缩到合适的目录，作者的习惯是在家目录建立一个 devel 目录，并将所有开发相关的软件都放到 devel 目录下：

```
cd ~/devel
tar xzvf ~/downloads/apache-mina-2.0.16-bin.tar.gz
```

我们看到在 apache-mina-2.0.16 目录下的文件如下：

```
$ tree -L 1
.
├── dist
├── docs
├── lib
├── LICENSE.jzlib.txt
├── LICENSE.ognl.txt
├── LICENSE.slf4j.txt
├── LICENSE.springframework.txt
├── LICENSE.txt
└── NOTICE.txt
```

其中的三个目录分别为：

- **dist**: mina 的发布文件，提供了 mina 的功能实现，我们下一步需要的 mina-core 就在这个目录下。
- **docs**: mina 的 API 文档。
- **lib**: mina 运行所需要支撑库。

---

<sup>8</sup>mina 的官方文档非常详尽，有需要详细了解 mina 的读者建议直接阅读 mina 的官方指南：<http://mina.apache.org/mina-project/userguide/user-guide-toc.html>，中文译本请参考：<https://waylau.gitbooks.io/apache-mina-2-user-guide/>

**Idea 中建立 mina 的开发环境** 在 Idea 中开发 mina 应用，需要将 mina 的核心库加入到项目的配置中，参见图5.8

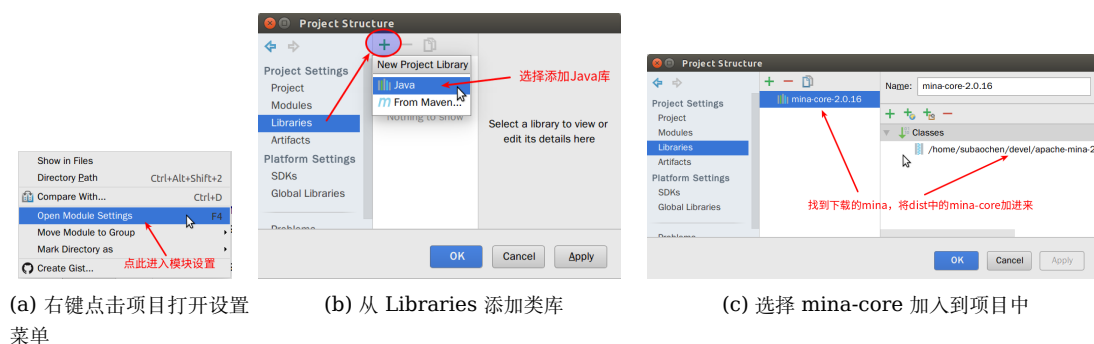


图 5.8: Idea 项目增加类库的方法

同样的方法，从 `apache-mina-2.0.16/lib` 目录下把 `slf4j-api-1.7.21.jar` 也加入到项目的库配置中。

### 5.3.3.2 简单 mina 应用程序示例

我们先从一个简单的“时间服务器”入手，看一下 mina socket server 是如何构建的。这个事件服务器实现的功能很简单，客户端无论发送什么消息上来，服务器只是返回当前时间作为响应。

如代码清单5.5所示，时间服务器的主类 `MinaTimeServer` 的主要工作是：

- 创建一个 `NioSocketAcceptor` 对象 `acceptor`，这是 mina 对使用 TCP 的 `SocketServer` 的进一步封装，我们此后可以利用这个 `acceptor` 完成 mina 服务器的配置和启动。
- 创建几个过滤器对象并添加到 `acceptor`。这里添加了两个过滤器：一是日志过滤器，实现对操作过程的日志处理；二是文本编码过滤器，使用 UTF-8 对 Socket 流中的字符进行编码转换。
- 设置这个时间服务器的事物逻辑处理器，即 `TimeServerHandler`，参见代码清单5.6。
- (可选) 设置 `acceptor` 监听器的配置属性，这里设置了缓冲器大小和 `idle` 时间。
- 将 `acceptor` 监听器绑定到指定的端口。

如代码清单5.6所示，在主类中我们使用 `TimeServerHandler` 来进行具体的业务逻辑处理，这里的逻辑很简单：当收到任何消息（即 `messageReceived` 方法被调用时）时，

我们创建新的 `Date` 对象并写入到发送队列 (`session.write()`)，mina 会将发送队列的消息返回到客户端。

代码清单 5.5: MinaTimeServer.java

```
1 package cn.edu.sdut.softlab.mina;
2
3 import org.apache.mina.core.service.IoAcceptor;
4 import org.apache.mina.core.session.IdleStatus;
5 import org.apache.mina.filter.codec.ProtocolCodecFilter;
6 import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
7 import org.apache.mina.filter.logging.LoggingFilter;
8 import org.apache.mina.transport.socket.nio.NioSocketAcceptor;
9
10 import java.io.IOException;
11 import java.net.InetSocketAddress;
12 import java.nio.charset.Charset;
13
14 /**
15  * Created by subaochen on 17-2-18.
16  */
17 public class MinaTimeServer {
18     private static final int PORT = 9123;
19
20     public static void main(String[] args) throws IOException {
21         // 创建服务器端的监听器对象
22         IoAcceptor acceptor = new NioSocketAcceptor();
23         // 增加日志过滤器：用于日志存储
24         acceptor.getFilterChain().addLast("logger", new LoggingFilter());
25         // 增加消息编码过滤器，采用UTF-8编码
26         acceptor.getFilterChain().addLast("codec",
27             new ProtocolCodecFilter(new TextLineCodecFactory(Charset.forName("UTF-8"))
28             ));
29         // 设置具体的事物逻辑处理器
30         acceptor.setHandler(new TimeServerHandler());
31         // 设置IoSession的一些属性
32         acceptor.getSessionConfig().setReadBufferSize(2048);
33         acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);
34         // 设置服务器监听的端口
35         acceptor.bind(new InetSocketAddress(PORT));
36     }
37 }
```

代码清单 5.6: TimeServerHandler.java

```
1 package cn.edu.sdut.softlab.mina;
2
3 import org.apache.mina.core.service.IoHandler;
4 import org.apache.mina.core.service.IoHandlerAdapter;
5 import org.apache.mina.core.session.IdleStatus;
6 import org.apache.mina.core.session.IoSession;
```

```
7
8 import java.util.Date;
9
10 /**
11  * Created by subaochen on 17-2-18.
12  */
13 public class TimeServerHandler extends IoHandlerAdapter {
14     @Override
15     public void exceptionCaught(IoSession session, Throwable cause) throws Exception {
16         cause.printStackTrace();
17     }
18
19     /**
20      * 当客户端收到服务器的消息后触发此方法。
21      * @param session IoSession对象
22      * @param message 客户端收到消息对象
23      * @throws Exception
24      */
25     @Override
26     public void messageReceived(IoSession session, Object message) throws Exception {
27         String str = message.toString();
28         if (str.trim().equalsIgnoreCase("quit")) {
29             session.close();
30             return;
31         }
32         Date date = new Date();
33         // 无论客户端发来什么消息，服务器只是把当前日期发送回去
34         session.write(date.toString());
35         System.out.println("Message written...");
36     }
37 }
```

运行 MinaTimeServer 后，我们可以通过 telnet 或者 socket 工具连接 MinaTimeServer，一个可能的运行结果如下：

```
$ telnet localhost 9123
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
Sun Feb 19 08:49:01 CST 2017
anybody there?
Sun Feb 19 08:49:33 CST 2017
```

### 5.3.3.3 mina 应用程序的基本结构

基于 Apache mina 的网络应用有三个层次,如图5.9<sup>9</sup>所示,分别是 I/O 服务、I/O

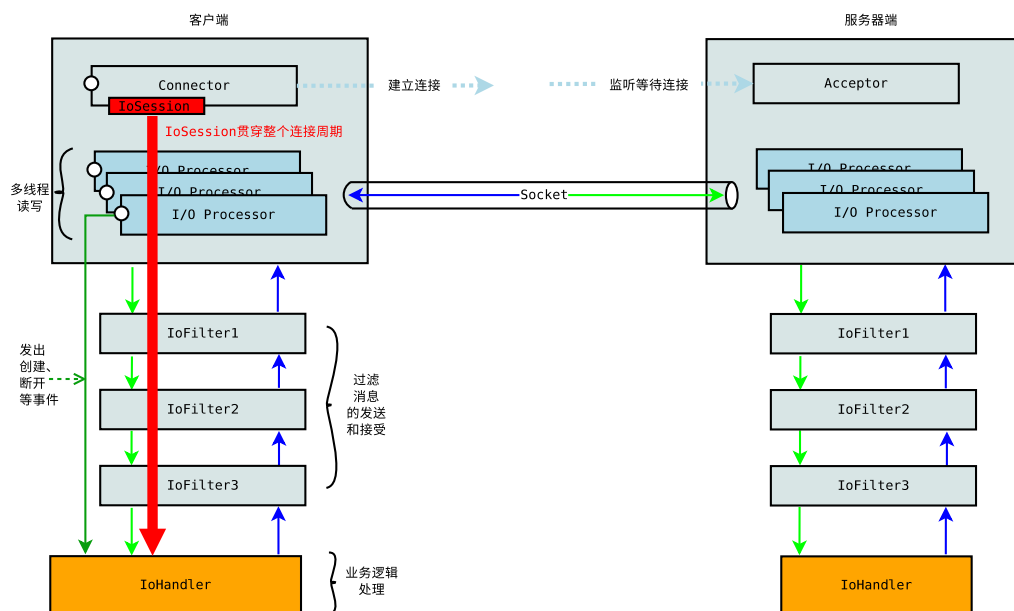


图 5.9: mina 框架各部分关系

过滤器和 I/O 处理器:

**I/O 服务 (I/O Processor) :** I/O 服务用来执行实际的网络 I/O 操作,即 Socket 通信。Apache mina 已经提供了一系列支持不同协议的 I/O 服务,如 TCP/IP、UDP/IP、串口和虚拟机内部的管道等,我们可以直接拿来使用。开发人员也可以实现 IoService 接口打造一个适合特定场景的 I/O 服务。

**I/O 过滤器 (IOFilter) :** I/O 服务能够传输的是字节流 (Socket 通讯),而上层应用 (即 IoHandler 部分) 需要的是特定的对象与数据结构。I/O 过滤器用来完成这两者之间的转换。I/O 过滤器的另外一个重要作用是对输入输出的数据进行处理,满足横切的需求。多个 I/O 过滤器串联起来,形成 I/O 过滤器链。mina 已经实现了常见的 IoFilter,我们可以根据需要进行选择使用,开发人员可以通过实现 IoFilter 接口或者扩展 IoFilterAdaptor 类打造特定的 I/O 过滤器。

**I/O 处理器 (Handler) :** I/O 处理器用来执行具体的业务逻辑,对接收到的消息执行特定的处理。通常,开发人员需要实现 IoHandler 接口或者扩展 IoHandlerAdaptor 类。

创建一个完整的基于 Apache mina 的网络应用,需要分别构建这三个层次。Apache mina 已经为 I/O 服务和 I/O 过滤器提供了不少的缺省实现,因此这两个层次在大多数

<sup>9</sup>本图借鉴了 <http://www.cnblogs.com/xuekyo/archive/2013/03/06/2945826.html>,感谢作者的精彩阐述。

情况下可以使用已有的实现。I/O 处理器由于是与具体的业务相关的，一般来说都是需要自己来实现的。

#### 5.3.3.4 事件驱动的异步 API

Apache mina 提供的是事件驱动的 API，它把与网络相关的各种活动抽象成事件，网络应用只需要对其感兴趣的事件进行处理即可。事件驱动的 API 使得基于 Apache mina 开发网络应用变得比较简单。应用不需要考虑与底层传输相关的具体细节，而只需要处理抽象的 I/O 事件。比如在实现一个服务端应用的时候，如果有新的连接进来，I/O 服务会产生 `sessionOpened` 这样一个事件。如果该应用需要在有连接打开的时候，执行某些特定的操作，只需要在 I/O 处理器中此事件处理方法 `sessionOpened` 中添加相应的代码即可。

**练习 5.3.** 使用 mina 编写一个计算算术表达式的服务器。

# 表 目 录

2.1	Collection 接口的方法 . . . . .	24
2.2	Set 接口的具体实现类 . . . . .	26
2.3	List 定义的额外方法 . . . . .	31
2.4	List 的具体实现类 . . . . .	32
2.5	Queue 的主要方法 . . . . .	38
2.6	Map 接口的方法 . . . . .	41
2.7	Iterator 的主要方法 . . . . .	46
3.1	Java8 新增的函数接口 . . . . .	62
3.2	Java8 的集合类 stream 方法 . . . . .	66
4.1	Thread 的常见方法 . . . . .	71
4.2	Object 中关于线程间通信的方法 . . . . .	90
5.1	URL 的构造方法 . . . . .	99



## 示例代码列表

1.1	Box.java . . . . .	4
1.2	GenericBox.java . . . . .	6
1.3	GenericMultiBox.java . . . . .	8
1.4	Generator.java . . . . .	10
1.5	FruitGenerator.java . . . . .	10
1.6	GenericMethodTest.java . . . . .	12
1.7	GenericArrayTest.java . . . . .	12
1.8	GenericBoxExtended.java . . . . .	14
1.9	GenericExtendsTest.java . . . . .	15
1.10	GenericExtendsComparableTest.java . . . . .	15
1.11	GenericSuperTest.java . . . . .	17
1.12	Check.java . . . . .	17
1.13	CheckTest.java . . . . .	18
2.1	SetNullTest.java . . . . .	26
2.2	FinDups.java . . . . .	27
2.3	FindDup2.java . . . . .	29
2.4	ListBasic.java . . . . .	32
2.5	Shuffle.java . . . . .	34
2.6	SubListTest.java . . . . .	36
2.7	CountDown.java . . . . .	39
2.8	WordFreq.java . . . . .	41
2.9	RefManager.java . . . . .	43
2.10	MapTranverse.java . . . . .	45
2.11	SetIteratorTest.java . . . . .	46
2.12	ListIteratorTest.java . . . . .	47
2.13	MapIterator.java . . . . .	49
2.14	extends 用法举例 . . . . .	52
2.15	super 用法举例 . . . . .	53
2.16	Collections.copy 方法 . . . . .	54

2.17	PECSTest.java	54
3.1	Student.java	57
3.2	LambdaScopeTest.java	68
4.1	RunnableThread.java	71
4.2	MyThread.java	72
4.3	MultiThreadDemo.java	72
4.4	Countdown.java	74
4.5	Countdown2.java	77
4.6	ThreadInterruptDemo.java	77
4.7	AnonThreadDemo.java	79
4.8	Counter.java	82
4.9	CounterClient.java	83
4.10	加锁属性的 Counter.java	85
4.11	UnsafeThreadDemo.java	87
4.12	SafeThreadLocalDemo.java	88
4.13	Buffer.java	90
4.14	Consumer.java	91
4.15	Producer.java	91
4.16	Client.java	92
5.1	ParseURL.java	100
5.2	URLReader.java	101
5.3	EchoServer.java	106
5.4	EchoClient.java	107
5.5	MinaTimeServer.java	111
5.6	TimeServerHandler.java	111