

第九章 集合框架

教学目标:

i 掌握集合框架的基本概念

i 掌握 Collection 接口

i 掌握 Iterator 接口

i 掌握 Set 接口

i 掌握 List 接口

i 掌握 Map 接口

i 熟悉集合的排序

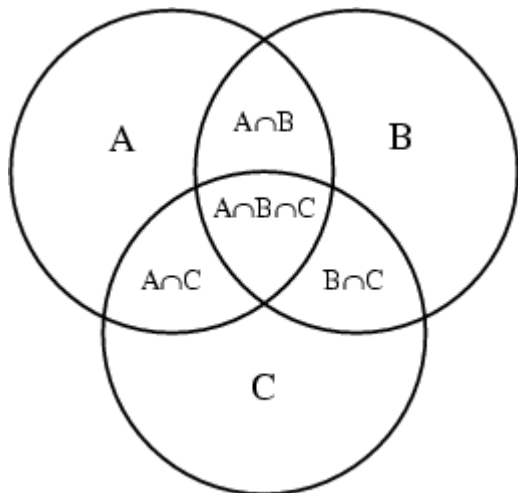
i 掌握泛型的概念

i 掌握泛型的编程应用

一：集合框架的基本概念

1: 数学背景

在常见用法中，集合 (*collection*) 和数学上直观的集 (*set*) 的概念是相同的。集是一个唯一项组，也就是说组中没有重复项。数学上集的概念比 Java 技术提前了一个世纪，那时英国数学家 George Boole 按逻辑正式的定义了集的概念。大部分人在小学时通过我们熟悉的维恩图引入的“集的交”和“集的并”学到过一些集的理论。



(1) 集的一些现实的示例如下：

- 大写字母集“A”到“Z”
- 非负整数集{0, 1, 2 ...}
- 保留的 Java 编程语言关键字集 {'import', 'class', 'public', 'protected' ...}
- 人集(friends, employees, clients, ...)
- 数据库查询返回记录集
- Container 的 Component 对象集
- 所有对 (pair) 集
- 空集{}

(2) 集的基本属性如下：

- 集内只包含每项的一个实例
- 集可以是有限的，也可以是无限的
- 可以定义抽象概念

(3) **映射**是一种特别的集。它是一种对 (pair) 集，每个对表示一个元素到另一元素的单向映射。一些映射示例有：

IP 地址到域名 (DNS) 的映射

关键字到数据库记录的映射

字典 (词到含义的映射)

2 进制到 10 进制转换的映射

此外，因为映射也是集，所以它们可以是有限的，也可以是无限的。无限映射的一个示例如 2 进制到 10 进制的转换。不幸的是，“集合框架”不支持无限映射——有时用数学函数、公式或算法更好。但在有限映射能解决问题时，“集合框架”会给 Java 程序员提供一个有用的 API。

2: 基本概念

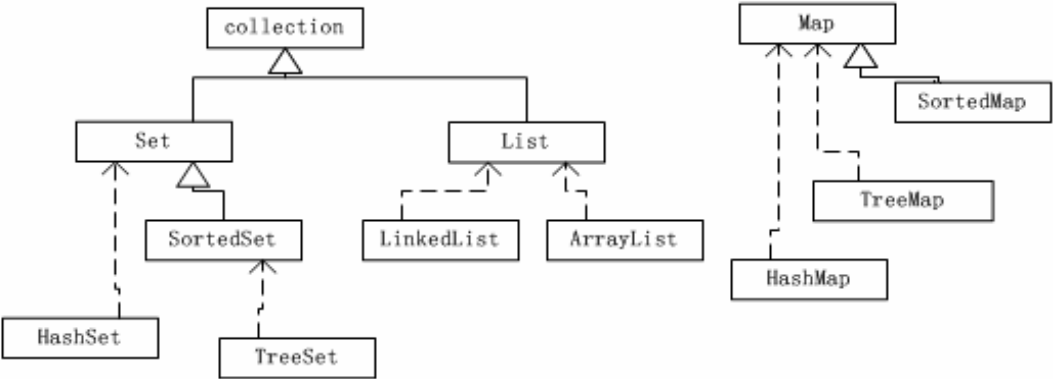
2.1: 什么是集合

集合是包含多个对象的简单对象，所包含的对象称为元素。

集合里面可以包含任意多个对象，数量可以变化；同时对对象的类型也没有限制，也就是说集合里面的所有对象的类型可以相同，也可以不同。

2.2: 集合里面有什么

既然您已经具备了一些集的理论，您应该能够更轻松的理解“集合框架”。“集合框架”由一组用来操作对象的接口组成。不同接口描述不同类型的组。虽然您总要创建接口特定的实现，但访问实际集合的方法应该限制在接口方法的使用上；因此，允许您更改基本的数据结构而不必改变其它代码。集合框架接口层次结构如下图所示。



有的人可能会认为 Map 会继承 Collection。在数学中，映射只是对 (pair) 的集合。但是，在“集合框架”中，接口 Map 和 Collection 在层次结构没有任何亲缘关系，它们是截然不同的。这种差别的原因与 Set 和 Map 在 Java 库中使用的方法有关。Map 的典型应用是访问按关键字存储的值。它支持一系列集合操作的全部，但操作的是键-值对，而不是单个独立的元素。因此 Map 需要支持 get() 和 put() 的基本操作，而 Set 不需要。此外，还有返回 Map 对象的 Set 视图的方法：Set set = aMap.keySet();

让我们转到对框架实现的研究，具体的集合类遵循命名约定，并将基本数据结构和框架接口相结合。除了四个历史集合类外，Java 框架还引入了六个集合实现，如下表所示。关于历史集合类如何转换、比如说，如何修改 Hashtable 并结合到框架中，请参阅历史集合类。

接口	实现	历史集合类
Set	HashSet	
	TreeSet	
List	ArrayList	Vector
	LinkedList	Stack
Map	HashMap	Hashtable
	TreeMap	Properties

这里没有 `Collection` 接口的实现。历史集合类，之所以这样命名是因为从 Java 类库 1.0 发行版就开始沿用至今了。

如果从历史集合类转换到新的框架类，主要差异之一在于所有的操作都和新类不同步。您可以往新类中添加同步的实现，但您不能把它从旧的类中除去。

2.3: 集合框架中各接口的特点

用“集合框架”设计软件时，记住该框架四个基本接口的下列层次结构关系会有用处：

`Collection` 接口是一组允许重复的对象。

`Set` 接口继承 `Collection`，无序但不允许重复。

`List` 接口继承 `Collection`，有序但允许重复，并引入位置下标。

`Map` 接口既不继承 `Set` 也不继承 `Collection`，是键值对。

二: `Collection` 接口

1: `Collection` 接口

`Collection` 接口用于表示任何对象或元素组。想要尽可能以常规方式处理一组元素时，就使用这一接口。这里是以统一建模语言（Unified Modeling Language (UML)）表示法表示的 `Collection` 公有方法清单。

<i>Collection</i>
+add(element : Object) : boolean +addAll(collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +hashCode() : int +iterator() : Iterator +remove(element : Object) : boolean +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +size() : int +toArray() : Object[] +toArray(array : Object[]) : Object[]

该接口支持如添加和除去等基本操作。设法除去一个元素时，如果这个元素存在，除去的仅仅是集合中此元素的一个实例。

`boolean add(Object element)`

`boolean remove(Object element)`

`Collection` 接口还支持查询操作：

`int size()`

`boolean isEmpty()`

`boolean contains(Object element)`

`Iterator iterator()`

2: `Iterator` 接口

`Collection` 接口的 `iterator()` 方法返回一个 `Iterator`。`Iterator` 和您可能已经熟悉的 `Enumeration` 接口类似，我们将在 `Enumeration` 接口中对 `Enumeration` 进行

讨论。使用 `Iterator` 接口方法,您可以从头至尾遍历集合,并安全的从底层 `Collection` 中除去元素:

<i>Iterator</i>
<code>+hasNext() : boolean</code>
<code>+next() : Object</code>
<code>+remove() : void</code>

`remove()` 方法可由底层集合有选择的支持。当底层集合调用并支持该方法时,最近一次 `next()` 调用返回的元素就被除去。为演示这一点,用于常规 `Collection` 的 `Iterator` 接口代码如下:

```
Collection collection = ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    Object element = iterator.next();
    if (removalCheck(element)) {
        iterator.remove();
    }
}
```

3: 组操作

`Collection` 接口支持的其它操作,要么是作用于元素组的任务,要么是同时作用于整个集合的任务。

```
boolean containsAll(Collection collection)
boolean addAll(Collection collection)
void clear()
void removeAll(Collection collection)
void retainAll(Collection collection)
```

`containsAll()` 方法允许您查找当前集合是否包含了另一个集合的所有元素,即另一个集合是否是当前集合的子集。其余方法是可选的,因为特定的集合可能不支持集合更改。

`addAll()` 方法确保另一个集合中的所有元素都被添加到当前的集合中,通常称为并。

`clear()` 方法从当前集合中除去所有元素。

`removeAll()` 方法类似于 `clear()`, 但只除去了元素的一个子集。

`retainAll()` 方法类似于 `removeAll()` 方法,不过可能感到它所做的与前面正好相反:它从当前集合中除去不属于另一个集合的元素,即交。

三: Set 接口

1: Set 接口

按照定义, `Set` 接口继承 `Collection` 接口,而且它不允许集合中存在重复项。所有原始方法都是现成的,没有引入新方法。具体的 `Set` 实现类依赖添加的对象的 `equals()` 方法来检查等同性。

Set
<pre>+add(element : Object) : boolean +addAll(collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +hashCode() : int +iterator() : Iterator +remove(element : Object) : boolean +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +size() : int +toArray() : Object[] +toArray(array : Object[]) : Object[]</pre>

2: HashSet 类和 TreeSet 类

“集合框架”支持 Set 接口两种普通的实现: HashSet 和 TreeSet。在更多情况下, 您会使用 HashSet 存储重复自由的集合。考虑到效率, 添加到 HashSet 的对象需要采用恰当分配散列码的方式来实现 hashCode() 方法。虽然大多数系统类覆盖了 Object 中缺省的 hashCode() 实现, 但创建您自己的要添加到 HashSet 的类时, 别忘了覆盖 hashCode()。当您要从集合中以有序的方式抽取元素时, TreeSet 实现会有用处。为了能顺利进行, 添加到 TreeSet 的元素必须是可排序的。

“集合框架”添加对 Comparable 元素的支持, 在排序的“可比较的接口”部分中会详细介绍。我们暂且假定一棵树知道如何保持 java.lang 包装程序器类元素的有序状态。一般说来, 先把元素添加到 HashSet, 再把集合转换为 TreeSet 来进行有序遍历会更快。

为优化 HashSet 空间的使用, 您可以调优初始容量和负载因子。TreeSet 不包含调优选项, 因为树总是平衡的, 保证了插入、删除、查询的性能为 $\log(n)$ 。

HashSet 和 TreeSet 都实现 Cloneable 接口。

3: 集的使用示例

为演示具体 Set 类的使用, 下面的程序创建了一个 HashSet, 并往里添加了一组名字, 其中有个名字添加了两次。接着, 程序把集中名字列表打印出来, 演示了重复的名字没有出现。接着, 程序把集作为 TreeSet 来处理, 并显示有序的列表。

```
import java.util.*;
```

```
public class SetExample {
    public static void main(String args[]) {
        Set set = new HashSet();
        set.add("Bernadine");
        set.add("Elizabeth");
        set.add("Gene");
        set.add("Elizabeth");
        set.add("Clara");
        System.out.println(set);
    }
}
```

```
        Set sortedSet = new TreeSet(set);
        System.out.println(sortedSet);
    }
}
```

运行程序产生了以下输出。请注意重复的条目只出现了一次, 列表的第二次输出已按字母顺序排序。

```
[Gene, Clara, Bernadine, Elizabeth]
[Bernadine, Clara, Elizabeth, Gene]
```

四: List 接口

1: List 接口

List 接口继承了 Collection 接口以定义一个允许重复项的有序集合。该接口不但能够对列表的一部分进行处理, 还添加了面向位置的操作。

<i>List</i>
<div><div>+add(element : Object) : boolean</div><div>+add(index : int, element : Object) : void</div><div>+addAll(collection : Collection) : boolean</div><div>+addAll(index : int, collection : Collection) : boolean</div><div>+clear() : void</div><div>+contains(element : Object) : boolean</div><div>+containsAll(collection : Collection) : boolean</div><div>+equals(object : Object) : boolean</div><div>+get(index : int) : Object</div><div>+hashCode() : int</div><div>+indexOf(element : Object) : int</div><div>+iterator() : Iterator</div><div>+lastIndexOf(element : Object) : int</div><div>+listIterator() : ListIterator</div><div>+listIterator(startIndex : int) : ListIterator</div><div>+remove(element : Object) : boolean</div><div>+remove(index : int) : Object</div><div>+removeAll(collection : Collection) : boolean</div><div>+retainAll(collection : Collection) : boolean</div><div>+set(index : int, element : Object) : Object</div><div>+size() : int</div><div>+subList(fromIndex : int, toIndex : int) : List</div><div>+toArray() : Object[]</div><div>+toArray(array : Object[]) : Object[]</div></div>

面向位置的操作包括插入某个元素或 Collection 的功能, 还包括获取、除去或更改元素的功能。在 List 中搜索元素可以从列表的头部或尾部开始, 如果找到元素, 还将报告元素所在的位置。

```
void add(int index, Object element)
boolean addAll(int index, Collection collection)
```

```
Object get(int index)
int indexOf(Object element)
int lastIndexOf(Object element)
Object remove(int index)
Object set(int index, Object element)
```

List 接口不但以位置友好的方式遍历整个列表, 还能处理集合的子集:

```
ListIterator listIterator()
ListIterator listIterator(int startIndex)
List subList(int fromIndex, int toIndex)
```

处理 subList() 时, 位于 fromIndex 的元素在子列表中, 而位于 toIndex 的元素则不是, 提醒这一点很重要。以下 for-loop 测试案例大致反映了这一点:

```
for (int i=fromIndex; i<toIndex; i++) {
    // process element at position i
}
```

此外, 我们还应该提醒的是 — 对子列表的更改 (如 add()、remove() 和 set() 调用) 对底层 List 也有影响。

2: ListIterator 接口

ListIterator 接口继承 Iterator 接口以支持添加或更改底层集合中的元素, 还支持双向访问。

<i>ListIterator</i>
+add(element: Object): void
+hasNext(): boolean
+hasPrevious(): boolean
+next(): Object
+nextIndex(): int
+previous(): Object
+previousIndex(): int
+remove(): void
+set(element: Object): void

以下源代码演示了列表中的反向循环。请注意 ListIterator 最初位于列表尾之后 (list.size()), 因为第一个元素的下标是 0。

```
List list = ...;
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
    Object element = iterator.previous();
    // Process element
}
```

正常情况下, 不用 ListIterator 改变某次遍历集合元素的方向 — 向前或者向后。

虽然在技术上可能实现时,但在 `previous()` 后立刻调用 `next()`, 返回的是同一个元素。把调用 `next()` 和 `previous()` 的顺序颠倒一下, 结果相同。

我们还需要稍微再解释一下 `add()` 操作。添加一个元素会导致新元素立刻被添加到隐式光标的前面。因此, 添加元素后调用 `previous()` 会返回新元素, 而调用 `next()` 则不起作用, 返回添加操作之前的下一个元素。

3: ArrayList 类和 LinkedList 类

在“集合框架”中有两种常规的 `List` 实现: `ArrayList` 和 `LinkedList`。使用两种 `List` 实现的哪一种取决于您特定的需要。如果要支持随机访问, 而不必在除尾部的任何位置插入或删除元素, 那么, `ArrayList` 提供了可选的集合。但如果, 您要频繁的从列表的中间位置添加和删除元素, 而只要顺序的访问列表元素, 那么, `LinkedList` 实现更好。

`ArrayList` 和 `LinkedList` 都实现 `Cloneable` 接口。此外, `LinkedList` 添加了一些处理列表两端元素的方法 (下图只显示了新方法):

LinkedList
+addFirst(element: Object): void
+addLast(element: Object): void
+getFirst(): Object
+getLast(): Object
+removeFirst(): Object
+removeLast(): Object

使用这些新方法, 您就可以轻松的把 `LinkedList` 当作一个堆栈、队列或其它面向端点的数据结构。

```
LinkedList queue = ...;
queue.addFirst(element);
Object object = queue.removeLast();
LinkedList stack = ...;
stack.addFirst(element);
Object object = stack.removeFirst();
```

`Vector` 类和 `Stack` 类是 `List` 接口的历史实现。

4: List 的使用示例

下面的程序演示了具体 `List` 类的使用。第一部分, 创建一个由 `ArrayList` 支持的 `List`。填充完列表以后, 特定条目就得到了。示例的 `LinkedList` 部分把 `LinkedList` 当作一个队列, 从队列头部添加东西, 从尾部除去。

```
import java.util.*;

public class ListExample {
    public static void main(String args[]) {
        List list = new ArrayList();
```

```
list.add("Bernadine");
list.add("Elizabeth");
list.add("Gene");
list.add("Elizabeth");
list.add("Clara");
System.out.println(list);
System.out.println("2: " + list.get(2));
System.out.println("0: " + list.get(0));
LinkedList queue = new LinkedList();
queue.addFirst("Bernadine");
queue.addFirst("Elizabeth");
queue.addFirst("Gene");
queue.addFirst("Elizabeth");
queue.addFirst("Clara");
System.out.println(queue);
queue.removeLast();
queue.removeLast();
System.out.println(queue);
}
```

运行程序产生了以下输出。请注意, 与 Set 不同的是 List 允许重复。

[Bernadine, Elizabeth, Gene, Elizabeth, Clara]

2: Gene

0: Bernadine

[Clara, Elizabeth, Gene, Elizabeth, Bernadine]

[Clara, Elizabeth, Gene]

五: Map 接口

1: Map 接口

Map 接口不是 Collection 接口的继承。而是从自己的用于维护键-值关联的接口层次结构入手。按定义, 该接口描述了从不重复的键到值的映射。

<i>Map</i>
+clear(): void +containsKey(key: Object): boolean +containsValue(value: Object): boolean +entrySet(): Set +get(key: Object): Object +isEmpty(): boolean +keySet(): Set +put(key: Object, value: Object): Object +putAll(mapping: Map): void +remove(key: Object): Object +size(): int +values(): Collection

我们可以把这个接口方法分成三组操作: 改变、查询和提供可选视图。

改变操作允许您从映射中添加和除去键-值对。键和值都可以为 `null`。但是, 您不能把 `Map` 作为一个键或值添加给自身。

```
Object put(Object key, Object value)
Object remove(Object key)
void putAll (Map mapping)
void clear()
```

查询操作允许您检查映射内容:

```
Object get(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()
```

最后一组方法允许您把键或值的组作为集合来处理。

```
public Set keySet()
public Collection values()
public Set entrySet()
```

因为映射中键的集合必须是唯一的, 您用 `Set` 支持。因为映射中值的集合可能不唯一, 您用 `Collection` 支持。最后一个方法返回一个实现 `Map.Entry` 接口的元素 `Set`。

2: Map.Entry 接口

`Map` 的 `entrySet()` 方法返回一个实现 `Map.Entry` 接口的对象集合。集合中每个对象都是底层 `Map` 中一个特定的键-值对。

<i>Map.Entry</i>
<pre>+equals(object : Object) : boolean +getKey() : Object +getValue() : Object +hashCode() : int +setValue(value : Object) : Object</pre>

通过这个集合迭代, 您可以获得每一条目的键或值并对值进行更改。但是, 如果底层 `Map` 在 `Map.Entry` 接口的 `setValue()` 方法外部被修改, 此条目集就会变得无效, 并导致迭代器行为未定义。

3: HashMap 类和 TreeMap 类

“集合框架”提供两种常规的 `Map` 实现: `HashMap` 和 `TreeMap`。和所有的具体实现一样, 使用哪种实现取决于您的特定需要。在 `Map` 中插入、删除和定位元素, `HashMap` 是最好的选择。但如果您要按顺序遍历键, 那么 `TreeMap` 会更好。根据集合大小, 先把元素添加到 `HashMap`, 再把这种映射转换成一个用于有序键遍历的 `TreeMap` 可能更快。使用

HashMap 要求添加的键类明确定义了 hashCode() 实现。有了 TreeMap 实现, 添加到映射的元素一定是可排序的。

HashMap 和 TreeMap 都实现 Cloneable 接口。

Hashtable 类和 Properties 类是 Map 接口的历史实现。

4: 映射的使用示例

以下程序演示了具体 Map 类的使用。该程序对自命令行传递的词进行频率计数。HashMap 起初用于数据存储。后来, 映射被转换为 TreeMap 以显示有序的键列列表。

```
import java.util.*;

public class MapExample {
    public static void main(String args[]) {
        Map map = new HashMap();
        Integer ONE = new Integer(1);
        for (int i=0, n=args.length; i<n; i++) {
            String key = args[i];
            Integer frequency = (Integer)map.get(key);
            if (frequency == null) {
                frequency = ONE;
            } else {
                int value = frequency.intValue();
                frequency = new Integer(value + 1);
            }
            map.put(key, frequency);
        }
        System.out.println(map);
        Map sortedMap = new TreeMap(map);
        System.out.println(sortedMap);
    }
}
```

用 Bill of Rights 的第三篇文章的文本运行程序产生下列输出, 请注意有序输出看起来多么有用!

无序输出:

```
{prescribed=1, a=1, time=2, any=1, no=1, shall=1, nor=1, peace=1, owner=1,
soldier=1, to=1, the=2, law=1, but=1, manner=1, without=1, house=1, in=4,
by=1, consent=1, war=1, quartered=1, be=2, of=3}
```

有序输出:

```
{a=1, any=1, be=2, but=1, by=1, consent=1, house=1, in=4, law=1, manner=1,
no=1, nor=1, of=3, owner=1, peace=1, prescribed=1, quartered=1, shall=1,
soldier=1, the=2, time=2, to=1, war=1, without=1}
```

六: 集合排序

为了用“集合框架”的额外部分把排序支持添加到 Java SDK, 版本 1.2, 核心 Java 库作了许多更改。像 `String` 和 `Integer` 类如今实现 `Comparable` 接口以提供自然排序顺序。对于那些没有自然顺序的类、或者当您想要一个不同于自然顺序的顺序时, 您可以实现 `Comparator` 接口来定义您自己的。

为了利用排序功能, “集合框架”提供了两种使用该功能的接口: `SortedSet` 和 `SortedMap`。

1: Comparable 接口

在 `java.lang` 包中, `Comparable` 接口适用于一个类有自然顺序的时候。假定对象集合是同一类型, 该接口允许您把集合排序成自然顺序。

<i>Comparable</i>
+compareTo(element : Object) : int

`compareTo()` 方法比较当前实例和作为参数传入的元素。如果排序过程中当前实例出现在参数前, 就返回某个负值。如果当前实例出现在参数后, 则返回正值。否则, 返回零。这里不要求零返回值表示元素相等。零返回值只是表示两个对象排在同一个位置。

在 Java SDK, 版本 1.2 中有十四类实现 `Comparable` 接口。下表展示了它们的自然排序。虽然一些类共享同一种自然排序, 但只有相互可比的类才能排序。

类	排序
<code>BigDecimal</code> , <code>BigInteger</code> , <code>Byte</code> , <code>Double</code> , <code>Float</code> , <code>Integer</code> , <code>Long</code> , <code>Short</code>	按数字大小排序
<code>Character</code>	按 Unicode 值的数字大小排序
<code>CollationKey</code>	按语言环境敏感的字符串排序
<code>Date</code>	按年代排序
<code>File</code>	按系统特定的路径名的全限定字符的 Unicode 值排序
<code>ObjectStreamField</code>	按名字中字符的 Unicode 值排序
<code>String</code>	按字符串中字符 Unicode 值排序

String 的 compareTo() 方法的文档按词典的形式定义了排序。这意味着比较只是在文本中被数字化了的字符串值之间, 其中文本没有必要按所有语言的字母顺序排序。对于语言环境特定的排序, 通过 CollationKey 使用 Collator。

下面演示了通过 CollationKey 使用 Collator 进行语言环境特定的排序:

```
import java.text.*;
import java.util.*;

public class CollatorTest {
    public static void main(String args[]) {
        Collator collator = Collator.getInstance();
        CollationKey key1 = collator.getCollationKey("Tom");
        CollationKey key2 = collator.getCollationKey("tom");
        CollationKey key3 = collator.getCollationKey("thom");
        CollationKey key4 = collator.getCollationKey("Thom");
        CollationKey key5 = collator.getCollationKey("Thomas");

        Set set = new TreeSet();
        set.add(key1);
        set.add(key2);
        set.add(key3);
        set.add(key4);
        set.add(key5);
        printCollection(set);
    }
    static private void printCollection(
        Collection collection) {
        boolean first = true;
        Iterator iterator = collection.iterator();
        System.out.print("[");
        while (iterator.hasNext()) {
            if (first) {
                first = false;
            } else {
                System.out.print(", ");
            }
            CollationKey key = (CollationKey)iterator.next();
            System.out.print(key.getSourceString());
        }
        System.out.println(")");
    }
}
```

运行程序产生了以下输出。

[thom, Thom, Thomas, tom, Tom]

如果没有用 `Collator`，而是直接的存储名字，那么小写的名字会和大写的名字分开显示：

[Thom, Thomas, Tom, thom, tom]

创建您自己的类 `Comparable` 只是个实现 `compareTo()` 方法的问题。通常就是依赖几个数据成员的自然排序。您自己的类也应该覆盖 `equals()` 和 `hashCode()` 以确保两个相等的对象返回同一个散列码。

2: `Comparator` 接口

若一个类不能用于实现 `java.lang.Comparable`，您可以提供自己的 `java.util.Comparator` 行为。如果您不喜欢缺省的 `Comparable` 行为，您照样可以提供自己的 `Comparator`。

<i>Comparator</i>
+compare(element1 : Object, element2 : Object) : int +equals(object : Object) : boolean

`Comparator` 的 `compare()` 方法的返回值和 `Comparable` 的 `compareTo()` 方法的返回值相似。在此情况下，如果排序时第一个元素出现在第二个元素之前，则返回一个负值。如果第一个元素出现在后，那么返回一个正值。否则，返回零。与 `Comparable` 相似，零返回值不表示元素相等。一个零返回值只是表示两个对象排在同一位置。由 `Comparator` 用户决定如何处理。如果两个不相等的元素比较的结果为零，您首先应该确信那就是您要的结果，然后记录行为。

为了演示，您会发现编写一个新的忽略大小写的 `Comparator`，代替使用 `Collator` 进行语言环境特定、忽略大小写的比较会更容易。这样的一种实现如下所示：

```
class CaseInsensitiveComparator implements Comparator {
    public int compare(Object element1, Object element2) {
        String lowerE1 = ((String)element1).toLowerCase();
        String lowerE2 = ((String)element2).toLowerCase();
        return lowerE1.compareTo(lowerE2);
    }
}
```

因为每个类在某些地方都建立了 `Object` 子类，所以这不是您实现 `equals()` 方法的必要条件。实际上大多数情况下您不会去这样做。切记该 `equals()` 方法检查的是 `Comparator` 实现的等同性，不是处于比较状态下的对象。

`Collections` 类有个预定义的 `Comparator` 用于重用。调用 `Collections.reverseOrder()` 返回一个 `Comparator`，它对逆序实现 `Comparable` 接口的对象进行排序。

3: SortedSet 接口

“集合框架”提供了个特殊的 Set 接口: SortedSet, 它保持元素的有序顺序。

SortedSet
+comparator() : Comparator +first() : Object +headSet(toElement : Object) : SortedSet +last() : Object +subSet(fromElement : Object, toElement : Object) : SortedSet +tailSet(fromElement : Object) : SortedSet

该接口为集的子集和它的两端（即头和尾）提供了访问方法。当您处理列表的子集时，更改子集会反映到源集。此外，更改源集也会反映在子集上。发生这种情况的原因在于子集由两端的元素而不是下标元素指定。此外，如果 fromElement 是源集的一部分，它就是子集的一部分。但如果 toElement 是源集的一部分，它却不是子集的一部分。如果您想要一个特殊的高端元素（to-element）在子集中，您必须找到下一个元素。对于一个 String 来说，下一个元素是个附带空字符的同一个字符串（string+"\0"）。;

添加到 SortedSet 的元素必须实现 Comparable, 否则您必须给它的实现类的构造函数提供一个 Comparator: TreeSet（您可以自己实现接口。但是“集合框架”只提供这样一个具体的实现类。）

为了演示，以下示例使用 Collections 类中逆序的 Comparator。

```
Comparator comparator = Collections.reverseOrder();
Set reverseSet = new TreeSet(comparator);
reverseSet.add("Bernadine");
reverseSet.add("Elizabeth");
reverseSet.add("Gene");
reverseSet.add("Elizabeth");
reverseSet.add("Clara");
System.out.println(reverseSet);
```

运行程序产生了以下输出。

```
[Gene, Elizabeth, Clara, Bernadine]
```

因为集必须包含唯一的项，如果添加元素时比较两个元素导致了零返回值（通过 Comparable 的 compareTo() 方法或 Comparator 的 compare() 方法）那么新元素就没有添加进去。如果两个元素相等，那还好。但如果它们不相等的话，您接下来就应该修改比较方法，让比较方法和 equals() 方法一致。

使用先前的 CaseInsensitiveComparator 演示这一问题，产生了一个三元素集: thom、Thomas 和 Tom，而不是可能预期的五个元素。

```
Comparator comparator = new CaseInsensitiveComparator();
Set set = new TreeSet(comparator);
set.add("Tom");
```



```
set.add("tom");
set.add("thom");
set.add("Thom");
set.add("Thomas");
```

4: SortedMap 接口

“集合框架”提供了个特殊的 Map 接口: SortedMap, 它用来保持键的有序顺序。

SortedMap
<pre>+comparator() : Comparator +firstKey() : Object +headMap(toKey : Object) : SortedMap +lastKey() : Object +subMap(fromKey : Object, toKey : Object) : SortedMap +tailMap(fromKey : Object) : SortedMap</pre>

此接口为映射的子集包括两个端点提供了访问方法。除了排序是作用于映射的键以外, 处理 SortedMap 和处理 SortedSet 一样。“集合框架”提供的实现类是 TreeMap。

因为对于映射来说, 每个键只能对应一个值, 如果在添加一个键-值对时比较两个键产生了零返回值 (通过 Comparable 的 compareTo() 方法或通过 Comparator 的 compare() 方法), 那么, 原始键对应值被新的值替代。如果两个元素相等, 那还好。但如果不相等, 那么您就应该修改比较方法, 让比较方法和 equals() 的效果一致。

七: 泛型

1: 产生泛型的动机

可以在集合框架 (Collection framework) 中看到泛型的动机。例如, Map 类允许您向一个 Map 添加任意类的对象, 即使最常见的情况是在给定映射 (map) 中保存某个特定类型 (比如 String) 的对象。

因为 Map.get() 被定义为返回 Object, 所以一般必须将 Map.get() 的结果强制类型转换为期望的类型, 如下面的代码所示:

```
Map m = new HashMap();
m.put("key", "bag");
String s = (String) m.get("key");
```

要让程序通过编译, 必须将 get() 的结果强制类型转换为 String, 并且希望结果真的是一个 String。但是有可能某人已经在该映射中保存了不是 String 的东西, 这样的话, 上面的代码将会抛出 ClassCastException。

理想情况下, 您可能会得出这样一个观点, 即 m 是一个 Map, 它将 String 键映射到 String 值。这可以让您消除代码中的强制类型转换, 同时获得一个附加的类型检查层, 该检查层可以防止有人将错误类型的键或值保存在集合中。这就是泛型所做的工作。

那么上面的代码使用泛型后, 修改如下:

```
Map<String,String> m = new HashMap<String,String>();
m.put("key", "bag");
String s = m.get("key");
```

2: 什么是泛型

泛型 (Generic type 或者 generics) 是对 Java 语言的类型系统的一种扩展, 以支持创建可以按类型进行参数化的类。可以把类型参数看作是使用参数化类型时指定的类型的一个占位符, 就像方法的形式参数是运行时传递的值的占位符一样。

泛型为提高大型程序的类型安全和可维护性带来了很大的潜力。泛型与其他几个 Java 语言特性相互协作, 包括增强的 for 循环 (有时叫做 foreach 或者 for/in 循环)、枚举 (enumeration) 和自动装箱 (autoboxing)。

在定义泛型类或声明泛型类的变量时, 使用尖括号来指定形式类型参数, 称为**类型形参**, 在调用时传递的实际类型成为**类型实参**。类型形参与类型实参之间的关系类似于形式方法参数与实际方法参数之间的关系, 只是类型参数表示类型, 而不是表示值, 示例如下:

```
Map<String,String> m = new HashMap<String,String>();
m.put("key", "bag");
String s = m.get("key");
```

注意, 必须指定两次类型参数。一次是在声明变量 map 的类型时, 另一次是在选择 HashMap 类的参数化以便可以实例化正确类型的一个实例时。

再看一个示例:

```
Collection<String> col = new ArrayList<String>();
col.add("Javass");
```

3: 使用泛型的好处

(1) 类型安全

泛型的主要目标是提高 Java 程序的类型安全。通过知道使用泛型定义的变量的类型限制, 编译器可以在一个高得多的程度上验证类型假设。

Java 程序中的一种流行技术是定义这样的集合, 即它的元素或键是公共类型的, 比如 “String 列表” 或者 “String 到 String 的映射”。通过在变量声明中捕获这一附加的类型信息, 泛型允许编译器实施这些附加的类型约束。类型错误现在就可以在编译时被捕获了, 而不是在运行时当作 ClassCastException 展示出来。将类型检查从运行时挪到编译时有助于您更容易找到错误, 并可提高程序的可靠性。

(2) 消除强制类型转换

泛型的一个附带好处是, 消除源代码中的许多强制类型转换。这使得代码更加可读, 并且减少了出错机会。尽管减少强制类型转换可以降低使用泛型类的代码的罗嗦程度, 但是声明泛型变量会带来相应的罗嗦。比较下面两个代码例子。

该代码不使用泛型:

```
List li = new ArrayList();
li.put(new Integer(3));
Integer i = (Integer) li.get(0);
```

该代码使用泛型:

```
List<Integer> li = new ArrayList<Integer>();
li.put(new Integer(3));
Integer i = li.get(0);
```

在简单的程序中使用一次泛型变量不会降低罗嗦程度。但是对于多次使用泛型变量的大型程序来说, 则可以累积起来降低罗嗦程度。

(3) 潜在的性能收益

泛型为较大的优化带来可能。在泛型的初始实现中, 编译器将强制类型转换 (没有泛型

的话,程序员会指定这些强制类型转换)插入生成的字节码中。但是更多类型信息可用于编译器这一事实,为未来版本的 JVM 的优化带来可能。

4: 命名类型参数

在泛型中,类型参数推荐的命名约定是使用大写的单个字母。使用一个字母可以同现实中那些具有描述性的,长的实际变量名有所区别。使用大写字母要同变量命名规则一致,并且要区别于局部变量,方法参数,成员变量,而这些变量常常使用一个小写字母。集合类中,比如 java.util 中常常使用类型变量 E 代表“Element type”。T 和 S 常常用来表示范型变量名(好像使用 i 和 j 作为循环变量一样)。

对于常见的泛型模式,推荐的名称是:

- K —— 键,比如 Map 的键
- V —— 值,比如 List 和 Set 的内容,或者 Map 中的值
- E —— 异常类,或者集合元素类型
- T —— 泛型

注意: 当一个变量被声明为泛型时,只能被实例变量和方法调用(还有内嵌类型)而不能被静态变量和方法调用。原因很简单,参数化的泛型是一些实例。静态成员是被类的实例和参数化的类所共享的,所以静态成员不应该有类型参数和他们关联。

5: 泛型不是协变的

关于泛型的混淆,一个常见的来源就是假设它们像数组一样是协变的。如果 A 扩展 B,那么 A 的数组也是 B 的数组,并且完全可以在需要 B[] 的地方使用 A[]: 如下:

```
Integer[] intArray = new Integer[10];
```

```
Number[] numberArray = intArray;
```

上面的代码是有效的,因为一个 Integer 是一个 Number,因而一个 Integer 数组是一个 Number 数组。但是对于泛型来说则不然。下面的代码是无效的:

```
List<Integer> intList = new ArrayList<Integer>();
```

```
List<Number> numberList = intList; // invalid
```

其实泛型不是协变的, List<Number>不是 List<Integer>的父类型。

最初,大多数 Java 程序员觉得这缺少协变很烦人,或者甚至是“坏的(broken)”,但是之所以这样有一个很好的原因。如果可以将 List<Integer> 赋给 List<Number>,下面的代码就会违背泛型应该提供的类型安全:

```
List<Integer> intList = new ArrayList<Integer>();
```

```
List<Number> numberList = intList; // invalid
```

```
numberList.add(new Float(3.1415));
```

因为 intList 和 numberList 都是有别名的,如果允许的话,上面的代码就会让您将不是 Integers 的东西放进 intList 中。但是,您有一个更加灵活的方式来定义泛型。

假设您具有该方法:

```
void printList(List l) {  
    for (Object o : l){  
        System.out.println(o);  
    }  
}
```

编译通过,但是如果试图用 `List<Integer>` 调用它,则会得到警告。出现警告是因为,您将泛型 (`List<Integer>`) 传递给一个只承诺将它当作 `List` (所谓的原始类型) 的方法,这将破坏使用泛型的类型安全。

如果试图编写像下面这样的方法,那么将会怎么样?

```
void printList(List<Object> l) {  
    for (Object o : l){  
        System.out.println(o);  
    }  
}
```

它仍然不会通过编译,因为一个 `List<Integer>` 不是一个 `List<Object>`。这才真正烦人——现在您的泛型版本还没有普通的非泛型版本有用!

解决方案是使用类型通配符。

6: 类型通配符

把上面的例子修改如下:

```
void printList(List<?> l) {  
    for (Object o : l){  
        System.out.println(o);  
    }  
}
```

上面代码中的问号是一个类型通配符。它读作“问号”。`List<?>` 是任何泛型 `List` 的父类型,所以可以将 `List<Object>`、`List<Integer>` 或 `List<List<List<AnyObject>>>` 传递给 `printList()`。

引入了类型通配符,这让您声明 `List<?>` 类型的变量。您可以对这样的 `List` 做什么呢?非常方便,可以从中检索元素,但是不能添加元素。原因不是编译器知道哪些方法修改列表哪些方法不修改列表,而是(大多数)变化的方法比不变化的方法需要更多的类型信息。下面的代码则工作得很好:

```
List<Integer> li = new ArrayList<Integer>();  
li.add(new Integer(42));  
List<?> lu = li;  
System.out.println(lu.get(0));
```

为什么该代码能工作呢?对于 `lu`,编译器一点都不知道 `List` 的类型参数的值。但是编译器比较聪明,它可以做一些类型推理。在本例中,它推断未知的类型参数必须扩展 `Object`。(这个特定的推理没有太大的跳跃,但是编译器可以作出一些非常令人佩服的类型推理,所以它让您调用 `List.get()` 并推断返回类型为 `Object`。

另一方面,下面的代码不能工作:

```
List<Integer> li = new ArrayList<Integer>();  
li.add(new Integer(42));  
List<?> lu = li;  
lu.add(new Integer(43)); // error
```

在本例中,对于 `lu`,编译器不能对 `List` 的类型参数作出足够严密的推理,以确定将 `Integer` 传递给 `List.add()` 是类型安全的。所以编译器将不允许您这么做。

以免您仍然认为编译器知道哪些方法更改列表的内容哪些不更改列表内容,请注意下面的代码将能工作,因为它不依赖于编译器必须知道关于 `lu` 的类型参数的任何信息:

```
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(42));
List<?> lu = li;
lu.clear();
```

7: 泛型方法

通过在类的定义中添加一个形式类型参数列表,可以将类泛型化。方法也可以被泛型化,不管它们定义在其中的类是不是泛型化的。

泛型类可在多个方法签名间实施类型约束。在 `List<V>` 中,类型参数 `V` 出现在 `get()`、`add()`、`contains()` 等方法签名中。当创建一个 `Map<K, V>` 类型的变量时,您就在方法之间宣称一个类型约束。您传递给 `add()` 的值将与 `get()` 返回的值的类型相同。

类似地,之所以声明泛型方法,一般是因为您想要在该方法的多个参数之间宣称一个类型约束。例如,下面代码中的 `ifThenElse()` 方法,根据它的第一个参数的布尔值,它将返回第二个或第三个参数:

```
public <T> T ifThenElse(boolean b, T first, T second) {
    return b ? first : second;
}
```

注意,您可以调用 `ifThenElse()`,而不用显式地告诉编译器,您想要 `T` 的什么值。编译器不必显式地被告知 `T` 将具有什么值;它只知道这些值都必须相同。编译器允许您调用下面的代码,因为编译器可以使用类型推理来推断出,替代 `T` 的 `String` 满足所有的类型约束:

```
String s = ifThenElse(b, "a", "b");
```

类似地,您可以调用:

```
Integer i = ifThenElse(b, new Integer(1), new Integer(2));
```

但是,编译器不允许下面的代码,因为没有类型会满足所需的类型约束:

```
String s = ifThenElse(b, "pi", new Float(3.14));
```

为什么您选择使用泛型方法,而不是将类型 `T` 添加到类定义呢?(至少)有两种情况应该这样做:

- 当泛型方法是静态的时,这种情况下不能使用类类型参数。
- 当 `T` 上的类型约束对于方法真正是局部的时,这意味着没有在不同类的另一个方法签名中使用相同类型 `T` 的约束。通过使得泛型方法的类型参数对于方法是局部的,可以简化封闭类型的签名。

8: 有限制类型 (extends 新的含义)

在 Java 语言引入泛型之前, `extends` 关键字总是意味着创建一个新的继承自另一个类或接口的类或接口。引入泛型之后, `extends` 关键字有了另一个含意。将 `extends` 用在类型参数的定义中 (`Collection<T extends Number>`) 或者通配符类型参数中 (`Collection<? extends Number>`), 用来表示有限制类型, 限制前面的“`T`”或者“`?`”必须是 `Number` 类型或者 `Number` 类型的子类型。

当使用 `extends` 来指示类型参数限制时,不需要子类-父类关系,只需要子类型-父类型关系。还要记住,有限制类型不需要是该限制的严格子类型;也可以是该限制。换句话说,对于 `Collection<? extends Number>`, 您可以赋给它 `Collection<Number>`、`Collection<Integer>`、`Collection<Long>`、`Collection<Float>` 等等。

9: 类型与类

泛型的引入使得 Java 语言中的类型系统更加复杂。以前,该语言具有两种类型 —— 引用类型和基本类型。对于引用类型,类型和类的概念基本上可以互换,术语子类型和子类也可以互换。

随着泛型的引入,类型和类之间的关系变得更加复杂。`List<Integer>` 和 `List<Object>` 是不同的类型,但是却是相同的类。尽管 `Integer` 扩展 `Object`,但是 `List<Integer>` 不是 `List<Object>`,并且不能赋给 `List<Object>` 或者强制转换成 `List<Object>`。

另一方面,现在有了一个新的古怪的类型叫做 `List<?>`,它是 `List<Integer>` 和 `List<Object>` 的父类。并且有一个更加古怪的 `List<? extends Number>`。

类型层次的结构和形状也变得复杂得多。类型和类不再几乎是相同的东西了。

基本上可以这么说:一个类具有多种类型,而且数量是无限的。

练习实践课:

程序 1:

通讯录

需求: 将朋友信息定义为一个对象, 将对象加入 **List** 中并从 **List** 中显示。

目标:

- 1、List 基本用法;
- 2、对象的封装。

程序:

```
//: Friend.java
package com.useful.java.part7;

public class Friend {
    //名字
    private String name;
    //邮箱
    private String email;
    //手机号
    private String mobile;
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setMobile(String mobile) {
        this.mobile = mobile;
    }

    public String getMobile() {
        return mobile;
    }
}
```



```
}  
}
```

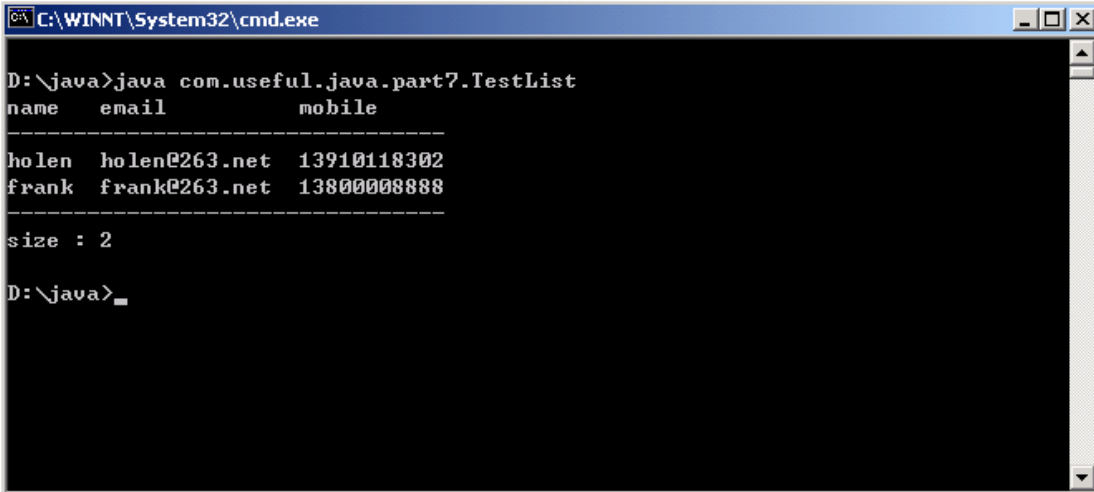
```
//: TestList.java  
package com.useful.java.part7;  
  
import java.util.*;  
  
public class TestList {  
    //friendList 是一个全局变量,用来存放所有的 Friend 对象  
    List friendList = new ArrayList();  
  
    public TestList() {  
    }  
  
    public void addFriend(Friend friend){  
        //调用 add 方法,往 list 中添加对象  
        friendList.add(friend);  
    }  
  
    public int getFriendSize(){  
        //size 是 list 对象的一个方法  
        return friendList.size();  
    }  
  
    public void insertFriend(String name,String email,String mobile){  
        //新建一个 Friend 对象  
        Friend friend = new Friend();  
        friend.setName(name);  
        friend.setEmail(email);  
        friend.setMobile(mobile);  
        this.addFriend(friend);  
    }  
  
    public void showAllFriend(){  
        System.out.println("name      " + "email      " + "mobile");  
        System.out.println("-----");  
        Friend friend = new Friend();  
        //循环显示,list 也是从 0 开始算起  
        for(int i = 0 ; i < this.getFriendSize(); i++){  
            friend = (Friend)friendList.get(i);  
            System.out.print(friend.getName() + " ");  
            System.out.print(friend.getEmail() + " ");  
            System.out.println(friend.getMobile() + " ");  
        }  
    }  
}
```



```
    }  
    System.out.println("-----");  
    System.out.println("size : " + this.getFriendSize());  
}  
  
public static void main(String[] args) {  
    TestList testList1 = new TestList();  
    //添加两条记录  
    testList1.insertFriend("holen","holen@263.net","13910118302");  
    testList1.insertFriend("frank","frank@263.net","13800008888");  
    testList1.showAllFriend();  
}  
}
```

说明:

- 1、List 主要用于有顺序的对象集合;
- 2、程序运行如下:



```
C:\WINNT\System32\cmd.exe  
D:\java>java com.useful.java.part7.TestList  
name    email      mobile  
-----  
holen   holen@263.net  13910118302  
frank   frank@263.net  13800008888  
-----  
size : 2  
D:\java>
```

- 3、Java 是面向对象的语言,在进行程序思维时,要注意增减有面向对象的思维方式。

程序 2:

无序的对象

需求: 将一些无序的对象置入一个对象集中。

目标:

- 1、HashTable 基本用法;
- 2、“键值对”的意义。

程序:

```
//: TestHashTable.java
package com.useful.java.part7;

import java.util.Hashtable;

public class TestHashTable {
    Hashtable hashTable = new Hashtable();
    String strObjectA = new String("this is ObjectA");
    String strObjectB = new String("This is ObjectB");
    Integer intObjectC = new Integer(2);
    Integer intObjectD = new Integer(2345);

    public TestHashTable() {
    }

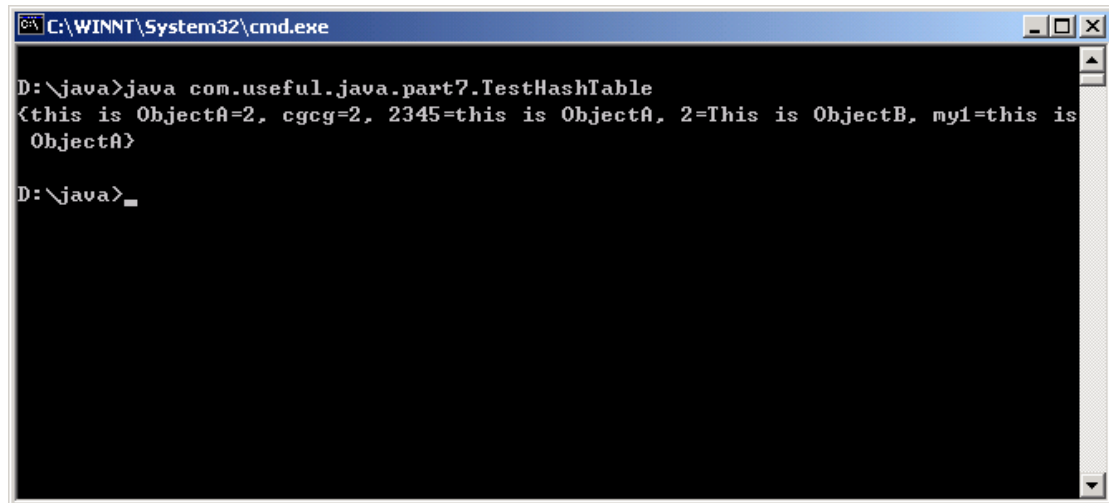
    public void addObject(){
        hashTable.put("my1",strObjectA);
        hashTable.put(intObjectC,strObjectB);
        hashTable.put(intObjectD,strObjectA);
        hashTable.put("cgcg",intObjectC);
        hashTable.put(strObjectA,intObjectC);
    }

    public void printHashtable(){
        System.out.println(hashTable);
    }

    public static void main(String[] args) {
        TestHashTable testHashTable1 = new TestHashTable();
        testHashTable1.addObject();
        testHashTable1.printHashtable();
    }
}
```

说明:

- 1、对于无序的对象集合，Hashtable、HashMap 将是不错的选择;
- 2、程序运行如下:



```
C:\WINNT\System32\cmd.exe

D:\java>java com.useful.java.part7.TestHashTable
{this is ObjectA=2, cgcg=2, 2345=this is ObjectA, 2=This is ObjectB, my1=this is
ObjectA}

D:\java>
```

作业

- 1: 定义一个类, 类里面有一个属性 col, 类型是集合类型 Collection, 实现下列方法: 可以向 col 里面添加数据、修改数据、查询数据、删除数据。也就是把这个 col 当作一个数据存储的容器, 对其实现数据的增删改查的方法。
- 2: 把上题的 Collection 改成使用 Map 实现

第十章 用户图形界面 GUI

教学目标:

- i 理解 GUI 的基本概念
- i 掌握 Frame、Panel 等容器的使用
- i 掌握 AWT 组件及使用
- i 掌握菜单的实现
- i 掌握如何控制外观
- i 熟悉布局管理器的使用

一: GUI 的基本概念

1: 什么是 GUI

GUI: Graphical User Interface , 图形用户接口, 即人机交互图形化用户界面, 就是屏幕产品的视觉体验和互动操作部分。经常读作“goo-ee”, 或者字母“G-U-I”, 或者完整的英文。切记, 不要按照汉语拼音读, 发出“贵”或者“鬼”的音, 闹笑话。

AWT (Abstract Window Toolkit: 抽象窗口工具箱), 包括了丰富的图形、用户界面组件和布局管理器的支持, 还有对界面事件的处理机制等。

2: Component

组件就是: 具有图形界面, 并能完成一定功能的封装体。

AWT 提供用于所有 Java 应用程序中的基本 GUI 组件, 还为应用程序提供与机器的界面。这将保证一台计算机上出现的東西与另一台上的相一致。

在学 AWT 之前, 简单回顾一下对象层次。记住, 超类是可以扩展的, 它们的属性是可继承的。而且, 类可以被抽象化, 这就是说, 它们是可被分成子类的模板, 子类用于类的具体实现。

显示在屏幕上的每个 GUI 组件都是抽象类组件的子类。也就是说, 每个从组件类扩展来的图形对象都与允许它们运行的大量方法和实例变量共享。

3: Container

容器就是: 用于包含其它组件的组件。

Container 是 Component 的一个抽象子类, 它允许其它的组件被嵌套在里面。这些组件也可以是允许其它组件被嵌套在里面的容器, 于是就创建了一个完整的层次结构。在屏幕上布置 GUI 组件, 容器是很有用的。Panel 是 Container 的最简单的类。Container 的另一个子类是 Window。

Window 是 Java.awt.Window 的对象。Window 是显示屏上独立的本机窗口, 它独立于其它容器。

Window 有两种形式: Frame(框架)和 Dialog (对话框)。Frame 和 Dialog 是 Window 的子类。Frame 是一个带有标题和缩放角的窗口。对话没有菜单条。尽管它能移动, 但它不能缩放。

Panel 是 Java.awt.Panel 的对象。Panel 包含在另一个容器中, 或是在 Web 浏览器的窗口中。Panel 确定一个四边形, 其它组件可以放入其中。Panel 必须放在 Window 之中 (或 Window 的子类中) 以便能显示出来。

注一容器不但能容纳组件, 还能容纳其它容器, 这一事实对于建立复杂的布局是关键的, 也是基本的。

4: 组件定位

容器里的组件的位置是由布局管理器决定的。容器对布局管理器的特定实例保持一个引用。当容器需要定位一个组件时, 它将调用布局管理器来做。当决定一个组件的大小时, 同样如此。布局管理器完全控制容器内的所有组件。它负责计算并定义上下文中对象在实际屏幕中所需的大小。

5: 组件大小

因为布局管理器负责容器里的组件的位置和大小,因此不需要总是自己去设定组件的大小或位置。如果想这样做(使用 `setLocation()`, `setSize()` 或 `setBounds()` 方法中的任何一种),布局管理器将覆盖你的决定。

如果必须控制组件的大小或位置,而使用标准布局管理器做不到,那就可能通过将下述方法调用发送到容器中来中止布局管理器: `setLayout(null)`;

做完这一步,必须对所有的组件使用 `setLocation()`, `setSize()` 或 `setBounds()`,来将它们定位在容器中。请注意,由于窗口系统和字体大小之间的不同,这种办法将导致从属于平台的布局。更好的途径是创建布局管理器的新子类。

6: 其它概念

在这里,再介绍几个其它的概念:

- (1): 界面是“画”出来的
- (2): 界面是“叠加”出来的,如果从侧面看,可以分做很多层
- (3): 常见的界面组合方式是:组件放在 `Panel` 里面, `Panel` 放在 `Frame` 上。也就是说,通常用 `Panel` 来进行界面的组合,一个 `Panel` 就是一个界面,那么界面的切换就相当于切换 `Frame` 里面的 `Panel`。

二: 容器的基本使用

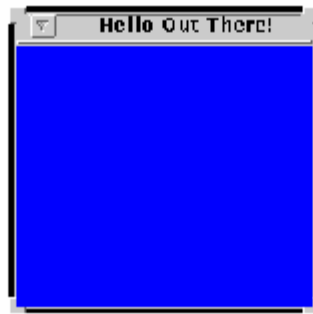
1: Frame 的使用

`Frames` 是 `Window` 的一个子类。它是带有标题和缩放角的窗口。它继承于 `Java.awt.Container`,因此,可以用 `add()` 方式来给框架添加组件。框架的缺省布局管理器就是 `Border Layout`。它可以用 `setLayout()` 方式来改变。

框架类中的构造程序 `Frame(String)` 用由 `String` 规定的标题来创建一个新的不可见的框架对象。当它还处于不可见状态时,将所有组件添加到框架中。

```
1. import java.awt.*;
2. public class MyFrame extends Frame {
3.     public static void main (String args[]) {
4.         MyFrame fr = new MyFrame("Hello Out There!");
5.         fr.setSize(500,500);
6.         fr.setBackground(Color.blue);
7.         fr.setVisible(true);
8.     }
9.     public MyFrame (String str) {
10.        super(str);
11.    }
12. }
```

上述程序创建了下述框架,它有一个具体的标题、大小及背景颜色。



注—在框架显示在屏幕上之前,必须做成可见的(通过调用程序 `setVisible(true)`),而且其大小是确定的(通过调用程序 `setSize()` 或 `pack()`)。

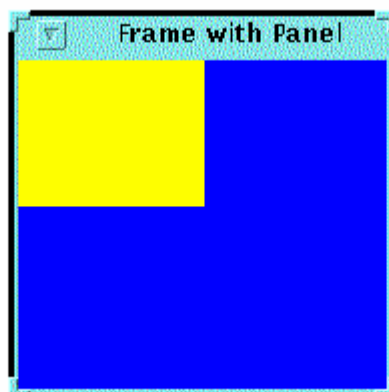
2: Panel 的使用

象 Frames 一样, Panels 提供空间来连接任何 GUI 组件,包括其它面板。每个面板都可以有它自己的布局管理程序。

一旦一个面板对象被创建,为了能看得见,它必须添加到窗口或框架对象上。用 Container 类中的 `add()` 方式可以做到这一点。

下面的程序创建了一个小的黄色面板,并将它加到一个框架对象上:

```
1. import java.awt.*;
2. public class FrameWithPanel extends Frame {
3.
4.     // Constructor
5.     public FrameWithPanel (String str) {
6.         super (str);
7.     }
8.
9.     public static void main (String args[]) {
10.        FrameWithPanel fr =
11.        new FrameWithPanel ("Frame with Panel");
12.        Panel pan = new Panel ();
13.
14.        fr.setSize(200, 200);
15.        fr.setBackground(Color. blue);
16.        fr.setLayout(null); //override default layout mgr
17.        pan.setSize (100, 100);
18.        pan.setBackground(Color. yellow);
19.
20.        fr.add(pan);
21.        fr.setVisible(true);
22.    }
23.    ....
```



三: AWT 组件的使用

AWT 组件提供了控制界面外观的机制, 包括用于文本显示的颜色和字体。此外, AWT 还支持打印。这个功能是在 JDK1.1 之后中引入的。

1: 按钮(Button)

你已经比较熟悉 Button 组件了。这个组件提供了“按下并动作”的基本用户界面。可以构造一个带文本标签的按钮, 用来告诉用户它的作用。

```
Button b = new Button("Sample");  
b.addActionListener(this);  
add(b);
```



任何实现了被注册为监听者的 ActionListener 接口的类, 它的 actionPerformed() 方法将在一个按钮被鼠标点击“按下”时被调用。

```
public void actionPerformed(ActionEvent ae) {  
    System.out.println("Button press received.");  
    System.out.println("Button's action command is: " +  
ae.getActionCommand());  
}
```

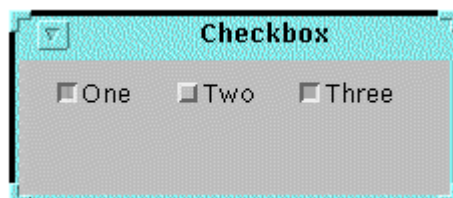
按钮被按下时调用的 getActionCommand() 方法在缺省情况下将返回标签字符串。用按钮的 setActionCommand() 方法改变动作命令和标签。


```
Button b = new Button("Sample");  
b.setActionCommand("Action Command Was Here!");  
b.addActionListener(this);  
add(b);
```

2: 复选框(Checkbox)

Checkbox 组件提供一种简单的“开/关”输入设备，它旁边有一个文本标签。

```
Frame f = new Frame("Checkbox")  
Checkbox one = new Checkbox("One", true);  
Checkbox two = new Checkbox("Two", false);  
Checkbox three = new Checkbox("Three", true);  
one.addItemListener(this);  
two.addItemListener(this);  
three.addItemListener(this);  
f.add(one);  
f.add(two);  
f.add(three);
```



选取或不选取(取消)一个复选框的事件将被送往 `ItemListener` 接口。所传递的 `ItemEvent` 包含 `getStateChange()` 方法，它根据实际情况返回 `ItemEvent.DESELECTED` 或 `ItemEvent.SELECTED`。`getItem()` 方法将受到影响的复选框作为一个表示这个复选框标签的 `String` 对象返回。

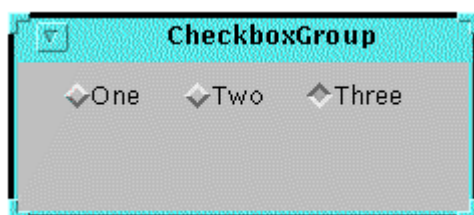
```
class Handler implements ItemListener {  
    public void itemStateChanged(ItemEvent ev) {  
        String state = "deselected";  
        if (ev.getStateChange() == ItemEvent.SELECTED){  
            state = "selected";  
        }  
    }  
}
```

```
    }  
  
    System.out.println(ev.getItem() + " " + state);  
  
    }  
  
}
```

3 复选框组—单选框(Checkbox group-Radio Button)

复选框组提供了将多个复选框作为互斥的一个集合的方法,因此在任何时刻,这个集合中只有一个复选框的值是 true。值为 true 的复选框就是当前被选中的复选框。你可以使用带有一个额外的 CheckboxGroup 参数的构造函数来创建一组中的每个复选框。正是这个 CheckBoxGroup 对象将各个复选框连接成一组。如果你这么做的话,那么复选框的外观会发生改变,而且所有和一个复选框组相关联的复选框将表现出“单选框”的行为。

```
Frame f = new Frame("Checkbox Group");  
  
CheckboxGroup cbg = new CheckboxGroup();  
  
Checkbox one = new Checkbox("One", false, cbg);  
Checkbox two = new Checkbox("Two", false, cbg);  
Checkbox three = new Checkbox("Three", true, cbg);  
  
one.addItemListener(this);  
two.addItemListener(this);  
three.addItemListener(this);  
  
f.add(one);  
f.add(two);  
f.add(three);
```



4 下拉列表(Choice)

下拉列表组件提供了一个简单的“从列表中选一个”类型的输入。例如:

```
Frame f = new Frame("Choice");  
  
Choice c = new Choice();  
  
c.add("First");
```

```
c.add("Second");  
c.add("Third");  
c.addItemListener(this);  
f.add(c);
```



点击下拉列表组件时, 它会显示一个列表, 列表中包含了所有加入其中的条目。注意所加入的条目是 String 对象。



ItemListener 接口用来观察下拉列表组件的变化, 其细节和复选框的相同。

5 画布(Canvas)

画布提供了一个空白(背景色)的空间。除非你用 `setSize()` 显式地定义它的大小, 否则它的大小就是 `0x0`。

画布的空间可以用来绘图、显示文本、接收键盘或鼠标的输入。后面的模块将告诉你如何有效地在 AWT 中绘图。

通常, 画布用来提供一个一般的绘图空间或者为客户组件提供工作区域。



画布可以“监听”所有适用于一个普通组件的事件。特别地, 你还可能想增加

KeyListener、MouseMotionListener 和 MouseListener 对象, 以允许某种方式对用户输入作出反应。

下面是画布的一个范例。每击一次键, 这个程序就改变一次画布的颜色。

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class MyCanvas extends Canvas
implements KeyListener {
    int index;
    Color colors[] = {Color.red, Color.green, Color.blue };
    public void paint(Graphics g) {
        g.setColor(colors[index]);
        g.fillRect(0,0,getSize().width,getSize().height);
    }

    public static void main(String args[]) {
        Frame f = new Frame("Canvas");
        MyCanvas mc = new MyCanvas();
        f.add(mc, BorderLayout.CENTER);
        f.setSize(150, 150);
        mc.requestFocus();
        mc.addKeyListener(mc);
        f.setVisible(true);
    }

    public void keyTyped(KeyEvent ev) {
        index++;
        if (index == colors.length) {
            index =0;
        }
        repaint();
    }

    public void keyPressed(KeyEvent ev) {
    }

    public void keyReleased(KeyEvent ev) {
    }
}
```

6 标签(Label)

一个标签对象显示一行静态文本。程序可以改变文本,但用户不能改变。标签没有任何特殊的边框和装饰。

```
Label l = new Label( " Hello " );  
add(l);
```

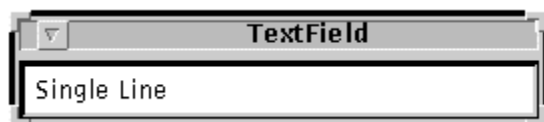


标签通常不处理事件,但也可以按照和画布相同的方式来处理事件。也就是说,只有调用了 `requestFocus()` 方法后,才能可靠地检取击键事件。

7 文本域(TextField)

文本域是一个单行的文本输入设备。例如:

```
TextField f = new TextField("Single line" , 30);  
f.addActionListener(this);  
add(f);
```



因为只允许有一行,所以当按下 Enter 或 Return 键时, `ActionListener` 可以通过 `actionPerformed()` 知道这个事件。如果需要,还可以增加其他的组件监听者。

除了注册一个 `ActionListener`,你还可以注册一个 `TextListener` 来接收关于个别击键的通知。它的回调方法是 `textValueChanged(TextEvent)`。

8 文本区(TextArea)

文本区是一个多行多列的文本输入设备。你可以用 `setEditable(boolean)` 将它设置成只读的。文本区将显示水平和垂直的滚动条。

下面这个范例创建了一个 4 行×30 字符的文本,最初它含有“Hello!”。

```
TextArea t = new TextArea( " Hello! " , 4, 30);  
t.addTextListener(this);  
add(t);
```



你用 `addTextListener` 指定的监听者将以和文本域相同的方式接收到关于击键的通知。

你可以给文本区增加一般的组件监听者,然而,由于文本是多行的,按下 `Enter` 键将导致把另一个字符送入缓冲。如果你需要识别“输入的结束”,你可以在文本区旁放置一个“应用”或“确认”按钮,以使用户指明“输入的结束”。

9 文本组件(Text Components)

文本区和文本域的文档都分为两个部分。如果你查找一个称为文本组件的类,你会找到若干个文本区和文本域共有的方法。例如,文本区和文本域都是文本组件的子类。

你已经知道文本区和文本域类的构造函数允许你指定显示所用的列数。记住所显示的组件大小是由布局管理器决定的,因此这些设置可能被忽略。进而,列数是按所用字体的平均宽度计算的。如果使用一种空间比例字体,实际显示的字符数可能相差很大。

由于文本组件实现了 `TextListener`,诸如文本域、文本区及其它子类都有对击键事件的内置支持。

10 列表(List)

一个列表将各个文本选项显示在一个区域中,这样就可以在同时看到若干个条目。列表可以滚动,并支持单选和多选两种模式。例如:

```
List l = new List(4, true);  
  
l.add("Hello");  
l.add("there");  
l.add("how");  
l.add("are");
```



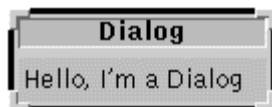
构造函数的数值参数定义了按可见列计算的列表高度。这个值也可能被布局管理器覆盖。一个值为 `true` 的布尔型参数表明这个列表允许用户作多个选择。



选取或取消一个条目时, AWT 将一个 `ItemEvent` 的实例送往列表。用户双击滚动列表中的一个条目时, 单选模式和多选模式的列表都会产生一个 `ActionEvent`。根据每个平台的约定来决定列表中的条目是否被选取。对于 UNIX/Motif 环境, 单击会加亮列表中的一个条目, 只有双击才会触发列表事件(选取条目)。

11: 对话框(Dialog)

对话框组件与一个框架相关联。它是一个带有一些装饰的自由窗口。它与框架的区别在于所提供的一些装饰, 而且你可以生成一个“模式”对话框, 它在被关闭前将存储所有窗口



的输入。

对话框可以是无模式和模式的。对于无模式对话框, 用户可以同时与框架和对话框交互。“模式”对话框在被关闭前将阻塞包括框架在内的其他所有应用程序的输入。

由于对话框是窗口的子类, 所以它的缺省布局管理器是 `Border Layout`。

```
Dialog d = new Dialog(f, "Dialog", false);  
  
d.add(new Label("Hello, I'm a Dialog", Border.Layout.CENTER));  
  
d.pack();
```

对话框在创建时通常是不可见的。通常在对按下按钮等用户输入作出反应时, 才显示对话框。

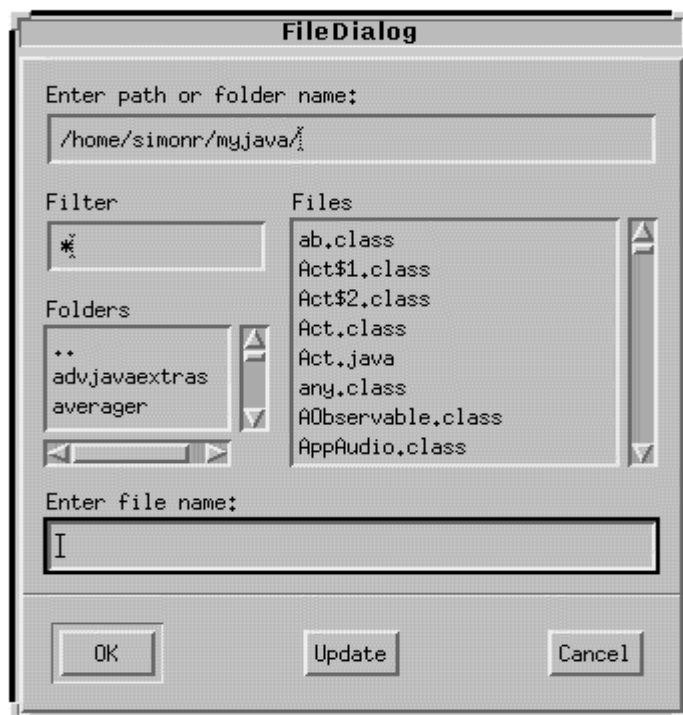
```
public void actionPerformed(ActionEvent ev) {  
  
    d.setVisible(true);  
  
}
```

要隐藏对话框, 你必须调用 `setVisible(false)`。典型的做法是对它添加一个 `WindowListener`, 并等待对那个监听者的 `windowClosing()` 调用。这和处理一个框架的关闭是平行的。

12: 文件对话框(File Dialog)

文件对话框是文件选择设备的一个实现。它有自己的自由窗口, 以及窗口元素, 并且允许用户浏览文件系统, 以及为以后的操作选择一个特定的文件。例如:

```
FileDialog d = new FileDialog(parentFrame, "FileDialog");  
  
d.setVisible(true); // block here until OK selected  
  
String fname = d.getFile();
```

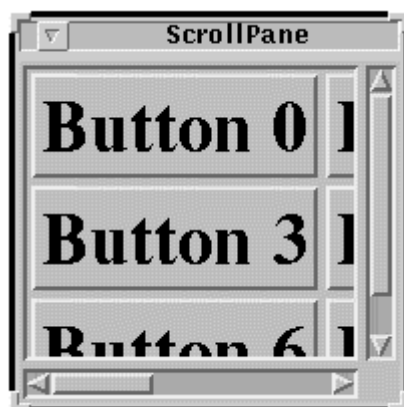


通常并不需要处理 FileDialog 的事件。调用 setVisible(true)将阻塞事件，直至用户选择 OK，这时会请求用户选择的文件名。这个信息将作为一个 String 返回。

13: 滚动面板(ScrollPane)

滚动面板提供了一种不能作为自由窗口的通用容器。它应当总是和一个容器相关联(例如，框架)。它提供了到一个更大的区域的视窗以及操纵这个视窗的滚动条。例如：

```
Frame f = new Frame("ScrollPane");
Panel p = new Panel();
ScrollPane sp = new ScrollPane();
p.setLayout(new GridLayout(3, 4));
sp.add(p);
f.add(sp, "Center");
f.setSize(200, 200);
f.setVisible(true);
```



滚动面板创建和管理滚动条,并持有一个组件。你不能控制它所用的布局管理器。你可以将一个面板加入到滚动面板中,配置面板的布局管理器,并在那个面板中放置你的组件。

通常,你不处理滚动面板上的事件;这些事件通过滚动面板所包含的组件进行处理。

四: 菜单的实现

菜单与其他组件有一个重要的不同:你不能将菜单添加到一般的容器中,而且不能使用布局管理器对它们进行布局。你只能将菜单加到一个菜单容器中。然而,你可以将一个 JMenuBar 组件加到一个 JContainer 中。你可以通过使用 setMenuBar() 方法将菜单放到一个框架中,从而启动一个菜单“树”。从那个时刻之后,你可以将菜单加到菜单条中,并将菜单或菜单项加到菜单中。

弹出式菜单是一个例外,因为它们可以以浮动窗口形式出现,因此不需要布局。

1: 帮助菜单

菜单条的一个特性是你可以将一个菜单指定为帮助菜单。这可以用 setHelpMenu(Menu) 来做到。要作为帮助菜单的菜单必须加入到菜单条中;然后它就会以和本地平台的帮助菜单同样的方式被处理。对于 X/Motif 类型的系统,这涉及将菜单条放置在菜单条的最右边。

2: 菜单条(MenuBar)

一个菜单条组件是一个水平菜单。它只能加入到一个框架中,并成为所有菜单树的根。在一个时刻,一个框架可以显示一个菜单条。然而,你可以根据程序的状态修改菜单条,这样在不同的时刻就可以显示不同的菜单。例如:

```
Frame f = new Frame("MenuBar");  
MenuBar mb = new MenuBar();  
f.setMenuBar(mb);
```



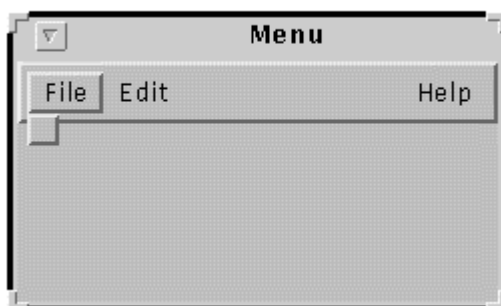
菜单条不支持监听者。作为普通菜单行为的一部分,在菜单条的区域中发生的预期事件会被自动处理。

3: 菜单组件

菜单组件提供了一个基本的下拉式菜单。它可以加入到一个菜单条或者另一个菜单中。例如:

```
MenuBar mb = new MenuBar();  
Menu m1 = new Menu("File");  
Menu m2 = new Menu("Edit");  
Menu m3 = new Menu("Help");
```

```
mb.add(m1);  
mb.add(m2);  
mb.setHelpMenu(m3);  
f.setMenuBar(mb);
```



注—这里显示的菜单是空的，这正是 File 菜单的外观。

你可以将一个 `ActionListener` 加入到菜单对象，但这种做法是罕见的。正常情况下，菜单用来显示和控制菜单条，这将在后面讨论。

4: 菜单项(MenuItem)

菜单项组件是菜单树的文本“叶”结点。它们通常被加入到菜单中，以构成一个完整的菜单。例如：

```
Menu m1 = new Menu("File");  
MenuItem mi1 = new MenuItem("New");  
MenuItem mi2 = new MenuItem("Load");  
MenuItem mi3 = new MenuItem("Save");  
MenuItem mi4 = new MenuItem("Quit");  
mi1.addActionListener(this);  
mi2.addActionListener(this);  
mi3.addActionListener(this);  
m1.add(mi1);  
m1.add(mi2);  
m1.addSeparator();  
m1.add(mi3);
```

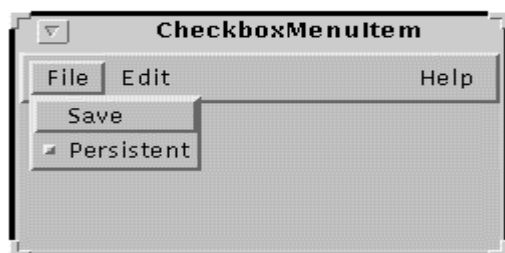
通常，将一个 `ActionListener` 加入到一个菜单项对象中，以提供菜单的行为。

5: 复选菜单项(CheckboxMenuItem)

复选菜单项是一个可复选的菜单项，可以在菜单上有选项(“开”或“关”)。例如：

```
Menu m1 = new Menu("File");  
MenuItem mi1 = new MenuItem("Save");  
CheckboxMenuItem mi2 =  
new CheckboxMenuItem("Persistent");  
mi1.addItemListener(this);  
mi2.addItemListener(this);  
m1.add(mi1);
```

```
m1.add(mi 2);
```



应当用 `ItemListener` 接口来监视复选菜单。因此当复选框状态发生改变时, 就会调用 `itemStateChanged()` 方法。

6: 弹出式菜单(PopupMenu)

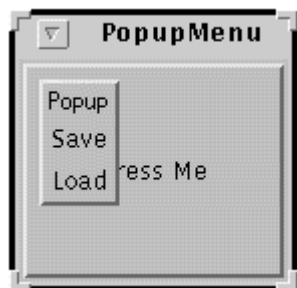
弹出式菜单提供了一种独立的菜单, 它可以在任何组件上显示。你可以将菜单条目和菜单加入到弹出式菜单中去。

例如:

```
Frame f = new Frame("PopupMenu");
Button b = new Button("Press Me");
PopupMenu p = new PopupMenu("Popup");
MenuItem s = new MenuItem("Save");
MenuItem l = new MenuItem("Load");
b.addActionListener(this);
f.add(b, BorderLayout.CENTER);
p.add(s);
p.add(l);
f.add(p);
```

为了显示弹出式菜单, 你必须调用显示方法。显示需要一个组件的引用, 作为 `x` 和 `y` 坐标轴的起点。通常, 你要为此使用组件的触发器。在上面这个范例中, 触发器是 `Button b`。
弹出式菜单 (续)

```
public void actionPerformed(ActionEvent ev) {
    p.show(b, 10, 10); // display popup
    // at (10,10) relative to b
}
```



注一弹出式菜单必须加入到一个“父”组件中。这与将组件加入到容器中是不同的。在

上面这个范例中, 弹出式菜单被加入到周围的框架中。

五: 控制外观

你可以控制在 AWT 组件中所显示的文本的前景背景颜色、背景颜色和字体。

1: 颜色

有两个方法用来设置组件的颜色:

- `setForeground()`
- `getForeground()`

这两个方法都带有一个参数, 参数是 `java.awt.Color` 类的实例。你可以使用常数颜色, 如 `Color.red`, `Color.blue` 等。所有预定义的颜色列在 `Color` 类的文档中。

此外, 你可以创建一个特定的颜色, 例如:

```
int r = 255, g = 255, b = 0;  
Color c = new Color(r, g, b);
```

上述构造函数根据指定的红色、绿色和蓝色的亮度(它们的范围都是 0~255)来创建一个颜色。

2: 字体

在组件中显示时所用的字体可以用 `setFont()` 方法来设置。这个方法的参数应当是 `java.awt.Font` 类的实例。

没有为字体定义常数, 但你可以根据字体的名称, 风格, 磅值来创建一个字体。

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

有效的字体名称包括:

- `Dialog`
- `Helvetica`
- `TimesRoman`
- `Courier`

可以通过调用 `Toolkit` 对象的 `getFontList()` 方法来得到完整的列表。`GetToolkit()` 方法是用来在显示 `toolkit` 后获得 `toolkit` 的。还有另外一种方法, 你可以使用缺省的 `toolkit`, 它可以通过调用 `Toolkit.getDefaultToolkit()` 来获得。

字体风格常数实际上是 `int` 值, 即:

- `Font.BOLD`
- `Font.ITALIC`
- `Font.PLAIN`
- `Font.BOLD + Font.ITALIC`

磅值必须使用 `int` 值来指定。

六: 布局管理器和布局

容器中组件的布局通常由布局管理器控制。每个 `Container` (比如一个 `Panel` 或一个 `Frame`) 都有一个与它相关的缺省布局管理器, 它可以通过调用 `setLayout()` 来改变。

布局管理器负责决定布局方针以及其容器的每一个子组件的大小。

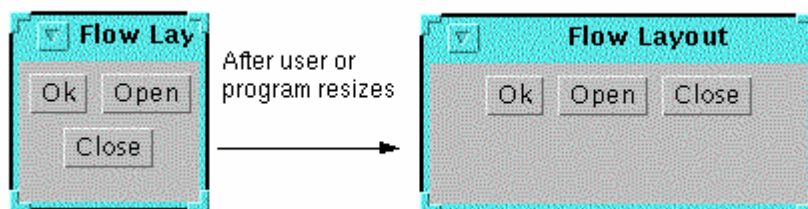
下面的布局管理器包含在 Java 编程语言中:

- Flow Layout—Panel 和 Applets 的缺省布局管理器
- Border Layout—Window、Dialog 及 Frame 的缺省管理程序
- Grid Layout
- Card Layout
- GridBag Layout

GridBag 布局管理器在本模块中不深入讨论。

1: Flow 布局管理器

Flow 布局管理器不限制它所管理的组件的大小, 而是允许它们有自己的最佳大小。如果想将组件设定缺省居中的话, Flow 布局构造程序参数允许将组件左对齐或右对齐。如果想在组件之间创建一个更大的最小间隔, 可以规定一个界限。当用户对由 Flow 布局管理的区域进行缩放时, 布局就发生变化。如:



下面的例子就是如何用类容器的 setLayout() 方法来创建 Flow 布局对象并安装它们。

```
setLayout(new FlowLayout(int align,int hgap, int vgap));
```

对齐的值必须是 FlowLayout.LEFT, FlowLayout.RIGHT, 或 FlowLayout.CENTER。例如:

```
setLayout(new FlowLayout(FlowLayout.RIGHT, 20, 40));
```

Flow Layout 的一个简单例子

```
import java.awt.*;

public class ExGui {
    private Frame f;
    private Button b1;
    private Button b2;

    public static void main(String args[]) {
        ExGui guiWindow = new ExGui();
        guiWindow.go();
    }

    public void go() {
```

```
f = new Frame("GUI example");
f.setLayout(new FlowLayout());
b1 = new Button("Press Me");
b2 = new Button("Don't Press Me");
f.add(b1);
f.add(b2);
f.pack();
f.setVisible(true);
}
}
```

main() 方法

本例中第 8 行 main() 方法有两个作用。首先, 它创建了 ExGui 对象的一个实例。回想一下, 直到一个实例存在, 还没有被称做 f, b1 和 b2 的真实数据项可以使用。第二, 当数据空间被创建时, main() 在该实例的上下文中调用实例方法 go()。在 go() 中, 真正的运行才开始。

new Frame ("GUI Example")

这个方法创建 Java.awt.Frame 类的一个实例。根据本地协议, 在 Java 编程语言中, Frame 是顶级窗口, 带有标题条—在这种情况下, 标题条由构造程序参数 "GUI Example" 定义—缩放柄, 以及其它修饰。

f.setLayout (new FlowLayout())

这个方法创建 Flow 布局管理器的一个实例, 并将它安装在框架中。对于每个 Frame、Border 布局来说, 都有一个布局管理器, 但本例中没有使用。Flow 布局管理器在 AWT 中是最简单的, 它在某种程度上象一个页面中的单词被安排成一行一行的那样来定位组件。请注意, Flow 布局缺省地将每一行居中。

new Button("Press Me")

这个方法创建 Java.awt.Button 类的一个实例。按钮是从本地窗口工具包中取出的一个标准按钮。按钮标签是由构造程序的字符串参数定义的。

f.add(b1)

这个方法告诉框架 f (它是一个容器), 它将包容组件 b1。b1 的大小和位置受从这一点向前的 Frame 布局管理器的控制。

f.pack()

这个方法告诉框架来设定大小, 能恰好密封它所包含的组件。为了确定框架要用多大, f.pack() 询问布局管理器, 在框架中哪个负责所有组件的大小和位置。

f.setVisible(true)

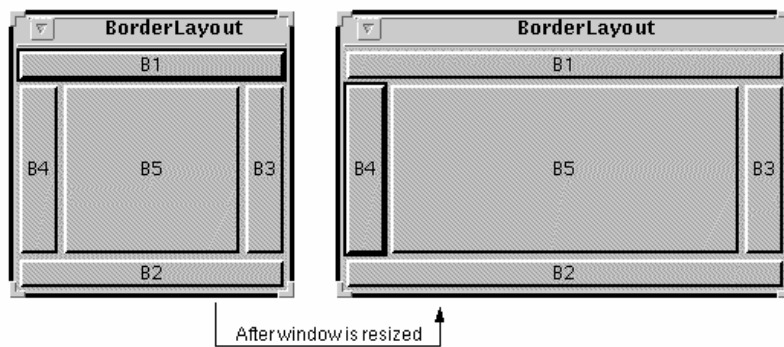
这个方法使框架以及其所有的内容变成用户看得见的东西。

2: Border 布局管理器

Border 布局管理器为在一个 Panel 或 Window 中放置组件提供一个更复杂的方案。Border 布局管理器包括五个明显的区域: 东、南、西、北、中。

北占据面板的上方, 东占据面板的右侧, 等等。中间区域是在东、南、西、北都填满后剩下的区域。当窗口垂直延伸时, 东、西、中区域也延伸; 而当窗口水平延伸时, 东、西、中区域也延伸。

Border 布局管理器是用于 Dialog 和 Frame 的缺省布局管理器。



注—当窗口缩放时，按钮相应的位置不变化，但其大小改变。

下面的代码对前例进行了修改，表示出了 Border 布局管理器的特性。可以用从 Container 类继承的 setLayout() 方法来将布局设定为 Border 布局。

```
import java.awt.*;

public class ExGui2 {
    private Frame f;
    private Button bn, bs, bw, be, bc;

    public static void main(String args[]) {
        ExGui2 guiWindow2 = new ExGui2();
        guiWindow2.go();
    }

    public void go() {
        f = new Frame("Border Layout");
        bn = new Button("B1");
        bs = new Button("B2");
        be = new Button("B3");
        bw = new Button("B4");
        bc = new Button("B5");

        f.add(bn, BorderLayout.NORTH);
        f.add(bs, BorderLayout.SOUTH);
        f.add(be, BorderLayout.EAST);
        f.add(bw, BorderLayout.WEST);
        f.add(bc, BorderLayout.CENTER);

        f.setSize(200, 200);
        f.setVisible(true);
    }
}
```


下面这一行:

```
setLayout(new BorderLayout());
```

构造并安装一个新 Border 布局, 在组件之间没有间隙。

这一行

```
setLayout(new BorderLayout(int hgap, int vgap));
```

构造并安装一个 Border 布局, 在由 hgap 和 vgap 规定的组件之间有规定的间隙。

在布局管理器中组件必须被添加到指定的区域, 而且还看不见。区域名称拼写要正确, 尤其是在选择不使用常量 (如 `add(button, "Center")`) 而使用 `add(button, BorderLayout.CENTER)` 时。拼写与大写很关键。

可以使用 Border 布局管理器来产生布局, 且带有在缩放时在一个方向、另一方向或双方向上都延伸的元素。

注—如果窗口水平缩放, 南、北、中区域变化; 如果窗口垂直缩放, 东、西、中区域变化;

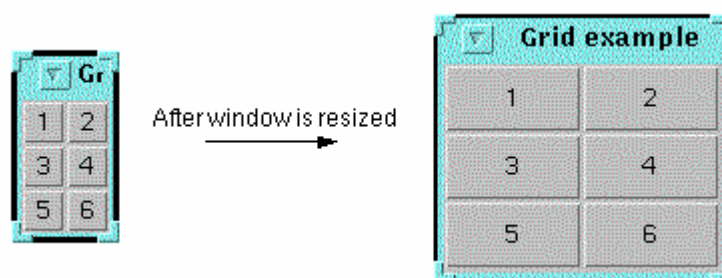
如果离开一个 Border 布局未使用的区域,, 好象它的大小为 0。中央区域即使在不含组件的情况下仍然呈现为背景。

可以仅将单个组件添加到 Border 布局管理器五个区域的每一个当中。如果添加不止一个, 只有最后一个看得见。后面的模块将演示如何用中间容器来允许不止一个组件被放在单个 Border 布局管理器区域的空间里。

注—布局管理器给予南、北组件最佳高度, 并强迫它们与容器一样宽。但对于东、西组件, 给予最佳宽度, 而高度受到限制。

3: Grid 布局管理器

Grid 布局管理器为放置组件提供了灵活性。用许多行和栏来创建管理程序。然后组件就填充到由管理程序规定的单元中。比如, 由语句 `new GridLayout(3, 2)` 创建的有三行两栏的 Grid 布局能产生如下六个单元:



因为有 Border 布局管理器, 组件相应的位置不随区域的缩放而改变。只是组件的大小改变。

Grid 布局管理器总是忽略组件的最佳大小。所有单元的宽度是相同的, 是根据单元数对可用宽度进行平分而定的。同样地, 所有单元的高度是相同的, 是根据行数对可用高度进行平分而定的。

将组件添加到网格中的命令决定它们占有的单元。单元的行数是从左到右填充, 就象文本一样, 而页是从上到下由行填充。


```
setLayout(new GridLayout());
```

创建并安装一个 Grid 布局, 每行中的每个组件有一个栏缺省。

```
setLayout(new GridLayout(int rows, int cols));
```

创建并安装一个带有规定好行数和栏数的 Grid 布局。对布局中所有组件所给的大小一样。

```
setLayout(new GridLayout(int rows, int cols, int hgap, int vgap);
```

创建并安装一个带有规定好行数和栏数的网格布局。布局中所有组件所给的大小一样。hgap 和 vgap 规定组件间各自的间隙。水平间隙放在左右两边及栏与栏之间。垂直间隙放在顶部、底部及每行之间。

注一行和栏中的一个, 不是两个同时, 可以为 0。这就是说, 任何数量的对象都可以放在一个行或一个栏中。

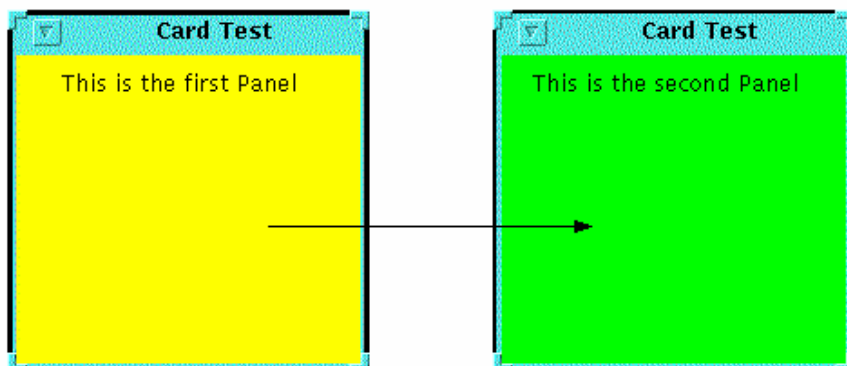
示例程序代码如下:

```
import java.awt.*;

public class GridEx {
    private Frame f;
    private Button b1, b2, b3, b4, b5, b6;
    public static void main(String args[]) {
        GridEx grid = new GridEx();
        grid.go();
    }
    public void go() {
        f = new Frame("Grid example");
        f.setLayout (new GridLayout (3, 2));
        b1 = new Button("1");
        b2 = new Button("2");
        b3 = new Button("3");
        b4 = new Button("4");
        b5 = new Button("5");
        b6 = new Button("6");
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.add(b6);
        f.pack();
        f.setVisible(true);
    }
}
```

4: Card 布局管理器

Card 布局管理器能将界面看作一系列的卡, 其中的一个在任何时候都可见。用 `add()` 方法来将卡添加到 Card 布局中。Card 布局管理器的 `show()` 方法应请求转换到一个新卡中。下例就是一个带有 5 张卡的框架。



鼠标点击左面板将视图转换到右面板。

用来创建上图框架的代码段如下所示:

```
import java.awt.*;
import java.awt.event.*;

public class CardTest implements MouseListener {

    Panel p1, p2, p3, p4, p5;
    Label l1, l2, l3, l4, l5;

    // Declare a CardLayout object,
    // to call its methods
    CardLayout myCard;
    Frame f;

    public static void main (String args[]) {
        CardTest ct = new CardTest ();
        ct.init();
    }

    public void init () {
        f = new Frame ("Card Test");
        myCard = new CardLayout();
        f.setLayout(myCard);

        // create the panels that I want
        // to use as cards
        p1 = new Panel ();
        p2 = new Panel ();
        p3 = new Panel ();
```

```
p4 = new Panel ();
p5 = new Panel ();
// create a label to attach to each panel, and
// change the color of each panel, so they are
// easily distinguishable

l1 = new Label ("This is the first Panel");
p1.setBackground(Color.yell ow);
p1.add(l1);

l2 = new Label ("This is the second Panel");
p2.setBackground(Color.green);
p2.add(l2);

l3 = new Label ("This is the third Panel");
p3.setBackground(Color.magenta);
p3.add(l3);

l4 = new Label ("This is the fourth Panel");
p4.setBackground(Color.whi te);
p4.add(l4);

l5 = new Label ("This is the fifth Panel");
p5.setBackground(Color.cyan);
p5.add(l5);

// Set up the event handling here ....
✓
// add each panel to my CardLayout
f.add(p1, "First");
f.add(p2, "Second");
f.add(p3, "Thi rd");
f.add(p4, "Fourth");
f.add(p5, "Fi fth");

// display the first panel
myCard.show(f, "First");

f.setSize (200, 200);
f.setVi sible(true);
}
```

5: GridBag 布局管理器

除了 Flow、Border、Grid 和 Card 布局管理器外, 核心 Java.awt 也提供 GridBag 布局管理器。

GridBag 布局管理器在网格的基础上提供复杂的布局,但它允许单个组件在一个单元中而不是填满整个单元那样地占用它们的最佳大小。网格包布局管理器也允许单个组件扩展成不止一个单元。

七: 谈谈 GUI 编程中的“画”界面

- 1: 首先是要建立一个 Application 的工程
- 2: 一个工程提供一个 Frame
- 3: 每个界面都是一个 Panel, 界面复杂程度取决于放在 Panel 里面组件的多少, 还有组件的复杂程度

所以, 画界面的过程基本上就是“画” Panel, 然后再 Panel 里面“画”组件的过程。

1: 一个 Panel 的代码的基本构成

经过分析, 常见的 Panel 类里面的代码分成如下 3 个部分:

- (1): 需要出现在界面上的组件的定义
- (2): 界面初始化的方法
- (3): 界面事件处理的方法

2: 组件的基本使用

分析在“画”界面的过程中, 组件的基本使用, 步骤如下:

- (1): 组件定义和组件初始化
- (2): 为组件设置相应的属性, 通常是组件外观的控制, 字体、颜色、大小、位置等
- (3): 把组件添加到 Panel, 把 Panel 添加到 Frame 中, 如果是 Frame, 那就显示出来

所有的组件基本上都是这么使用, 复杂的组件主要是在属性和方法上复杂, 但是使用的方法是差不多的。

练习实践

本章的内容为 AWT 实践重点:

- I AWT 基础
- I 界面布局

程序 1:

获取文件路径

需求: 选择一个文件, 获取其全路径。

目标:

- 1、打开文件对话框、保存文件对话框的使用;
- 2、内部类的使用;
- 3、如何处理点击事件。

程序:

```
//: FileDialogNew.java
package com.useful.java.part4;

import java.awt.*;
import java.awt.event.*;

public class FileDialogNew extends Frame {
    TextField filename = new TextField();
    TextField directory = new TextField();
    Button open = new Button("Open");
    Button save = new Button("Save");
    public FileDialogNew() {
        setTitle("File Dialog Test");
        Panel p = new Panel();
        p.setLayout(new FlowLayout());
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        add(p, BorderLayout.SOUTH);
        directory.setEditable(false);
        filename.setEditable(false);
        p = new Panel();
        p.setLayout(new GridLayout(2,1));
        p.add(filename);
        p.add(directory);
        add(p, BorderLayout.NORTH);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
```

```
// Two arguments, defaults to open file:
FileDialog d = new FileDialog(
    FileDialogNew.this,
    "What file do you want to open?");
d.setFile("*.java");
d.setDirectory("."); // Current directory
d.show();
String yourFile = ".*.*";
if((yourFile = d.getFile()) != null) {
    filename.setText(yourFile);
    directory.setText(d.getDirectory());
} else {
    filename.setText("You pressed cancel");
    directory.setText("");
}
}

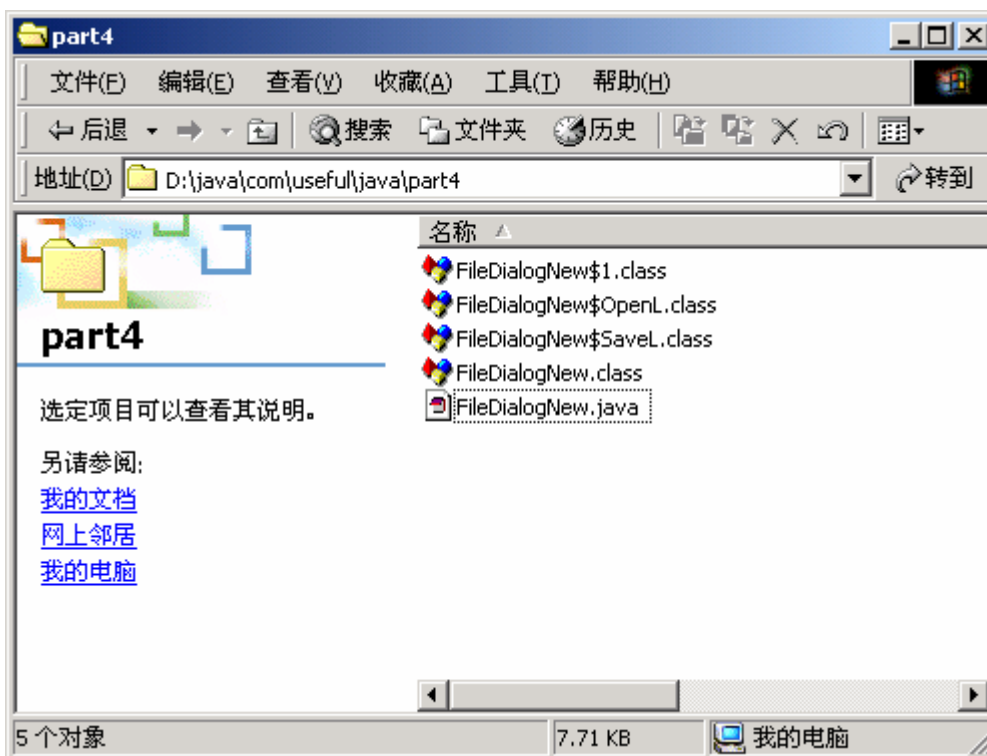
class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileDialog d = new FileDialog(
            FileDialogNew.this,
            "What file do you want to save?",
            FileDialog.SAVE);
        d.setFile("*.java");
        d.setDirectory(".");
        d.show();
        String saveFile;
        if((saveFile = d.getFile()) != null) {
            filename.setText(saveFile);
            directory.setText(d.getDirectory());
        } else {
            filename.setText("You pressed cancel");
            directory.setText("");
        }
    }
}

public static void main(String[] args) {
    Frame f = new FileDialogNew();
    f.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
}
```

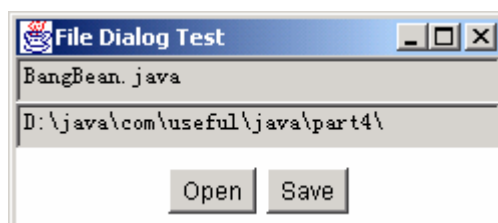
```
f.setSize(250,110);  
f.setVisible(true);  
  
}  
}
```

说明:

- 1、 程序中多次用到内部类, 在 GUI 程序经常用到内部类, 内部类编译后其类名由两部分组成, 前者为主类名, 后面是本类名, 中间用\$隔开, 如下图所示:



- 2、 事件处理时, 使用了接口 ActionListener, 这也是通用的事件处理方式之一;
- 3、 程序运行如下:



作业

- 1、 构建一个地址本的应用, 画出增、删、改、查、列表的界面

第十一章 Swing 和 GUI 事件处理

教学目标:

- i 理解 Java 基础类基本概念
- i 了解 Swing 包
- i 掌握 Swing 组件
- i 理解 Jcomponent 类
- i 掌握 GUI 事件的基本概念
- i 掌握如何编写代码来处理 GUI 事件
- i 熟悉 Adapter 类的编写和应用
- i 掌握如何编写 Swing 程序

一: Java 基础类 JFC 的基本概念

Java 基础类是关于 GUI 组件和服务的完整集合,它大大简化了健壮 Java 应用程序的开发和实现。JFC 作为 JDK1.2 的一个有机部分,主要包含 5 个 API: AWT, JavaD, Accessibility, Drag & Drop, Swing。它提供了帮助开发人员设计复杂应用程序的一整套应用程序开发包。

正如前面那些模块中所讨论的那样,AWT 组件为 Java 应用程序提供了多种 GUI 工具。

JavaD 是一图形 API,它为 Java 应用程序提供了一套高级的有关二维(2D)图形图像处理的类。JavaD API 扩展了 `java.awt` 和 `java.awt.image` 类,并提供了丰富的绘图风格,定义复杂图形的机制和精心调节绘制过程的方法和类。这些 API 使得独立于平台的图形应用程序的开发更加简便。

Accessibility API 提供了一套高级工具,用以辅助开发使用非传统输入和输出的应用程序。它提供了一个辅助的技术接口,如:屏幕阅读器,屏幕放大器,听觉文本阅读器(语音处理)等等。

Drag & Drop 技术提供了 Java 和本地应用程序之间的互操作性,用来在 Java 应用程序和不支持 Java 技术的应用程序之间交换数据。

JFC 模块的重点在 Swing。Swing 用来进行基于窗口的应用程序开发,它提供了一套丰富的组件和工作框架,以指定 GUI 如何独立于平台地展现其视觉效果。

二: Swing 包

1: Swing 介绍

Swing 提供了一整套 GUI 组件,为了保证可移植性,它是完全用 Java 语言编写的。

所有 Swing 都作为 JComponent 的子类来实现,而 JComponent 类又是从 Container 类继承而来。Swing 组件从 JComponent 继承了如下功能:

边框

你可以用 `setBorder()` 方法来指定在组件周围显示的边框。还可用一个 `EmptyBorder` 的实例来指定一个组件在其周围留有一定的额外空间。

双缓冲

双缓冲可以改善一个频繁被改变的组件的外观。现在你不需要编写双缓冲代码——Swing 已为你提供了。缺省情况下,Swing 组件是双缓冲的。

提示框

用 `setToolTipText()` 方法来指定一个字符串,你可以提供给用户有关某个组件的帮助信息。当光标暂停在组件上时,所指定的字符串就会在组件附近的一个小窗口中显示出来。

键盘导航

使用 `registerKeyboardAction()` 方法,你可以让用户以键盘代替鼠标来操作 GUI。用户为启动一个动作所必须按下的修饰键与字符的组合,由一个 `KeyStroke` 对象来表示。

应用程序范围的可插式外观和感觉

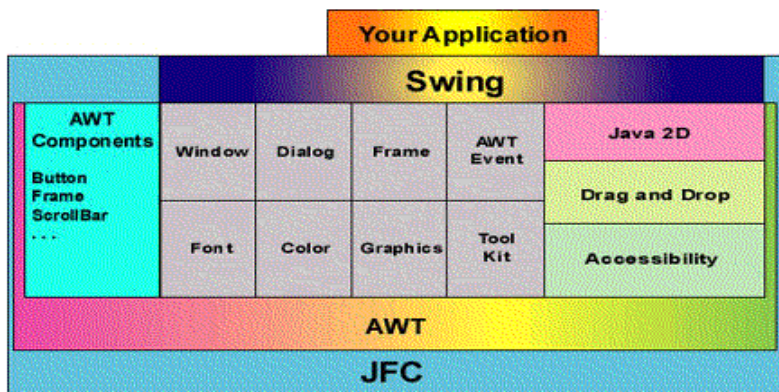
每个 Java 应用程序在运行时刻有一个 `GUIManager` 对象,它用于确定运行时刻 Swing 组件的外观和感觉。由于安全性的限制,你可以通过调用 `UIManager.setLookAndFeel()` 方法选择所有 Swing 组件的外观和感觉。在你所看见的东西背后,每个 JComponent 对象都有一个对应的 `ComponentGUI` 对象,它用来执行所有关于该 JComponent 的绘制、事件处理、大小判定等任务。

可插的外观和感觉使得开发人员可以构建这样的应用程序:它们可以在任何平台上执行,而且看上去就象是专门为那个特定的平台而开发的。开发人员可以创建自己的客户化 Swing 组件,带有他们想设计出的任何外观和感觉。这增加了用于跨平台应用程序和 Applet

的可靠性和一致性。一个完整应用程序的 GUI 可以在运行时刻从一种外观切换到另一种。

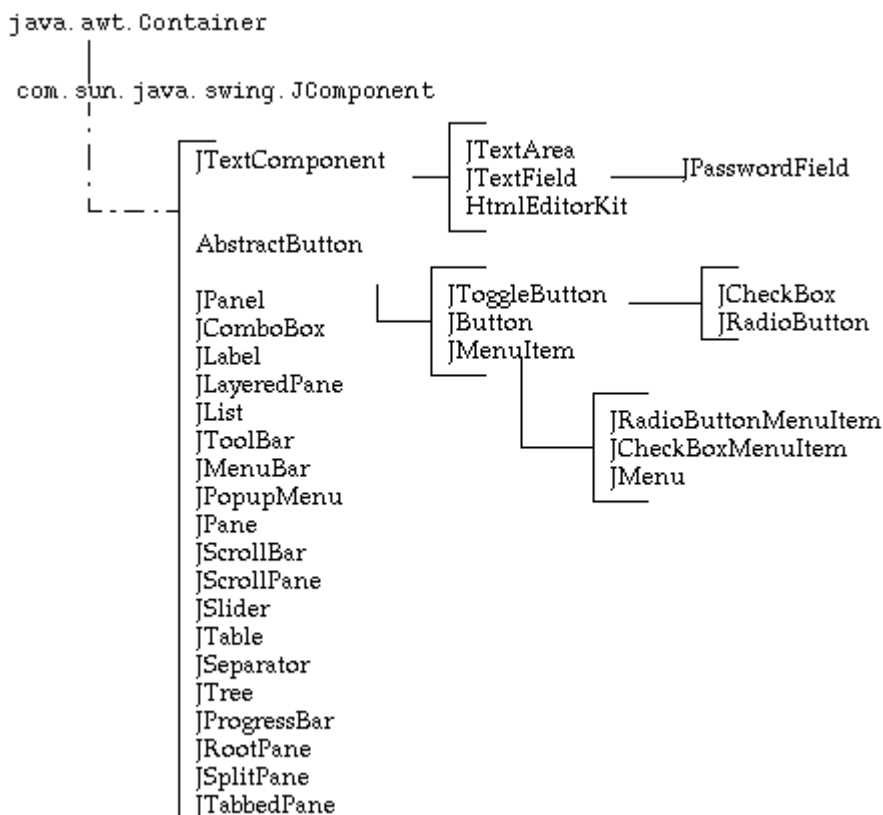
2: Swing 的体系结构

与 AWT 比较, Swing 提供了更完整的组件, 引入了许多新的特性和能力。Swing API 是围绕着实现 AWT 各个部分的 API 构筑的。这保证了所有早期的 AWT 组件仍然可以使用。AWT 采用了与特定平台相关的实现, 而绝大多数 Swing 组件却不是这样做的, 因此 Swing 的外观和感觉是可客户化和可插的。



上图显示了 JFC 各个部分之间的相互关系。JavaD, Accessibility, Drag & Drop, 和 Accessibility API 是 AWT 和 JFC 的一部分, 但它们不属于 Swing。这是因为, 这些组件使用了一些本地代码, 而 Swing 却不是这样的。

下图说明了 Swing 组件的层次结构:



Swing GUI 使用两种类型的类, 即 GUI 类和非 GUI 支持类。GUI 类是可视的, 它从 JComponent 继承而来, 因此称为“J”类。非 GUI 类为 GUI 类提供服务, 并执行关键功能; 因此它们不产生任何可视的输出。

三: Swing 组件

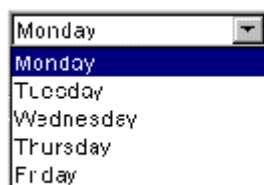
Swing 组件主要为文本处理、按钮、标签、列表、pane、组合框、滚动条、滚动 pane、菜单、表格和树提供了组件。其中一些组件如下所示:



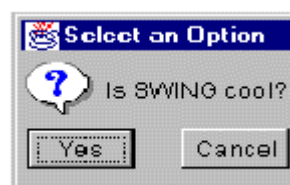
JApplet



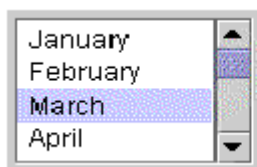
Jbutton



JComboBox



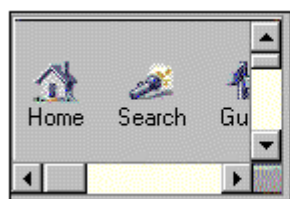
JOptionPane



JList



JLabel



JScrollPane

First Na..	Last Name
Mark	Andrews
Tom	Ball
Alar	Ching
Jeff	Cinkins

JTable



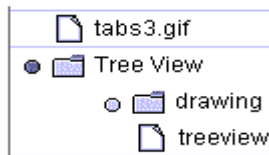
JScrollBar



JSlider



JTooltip



JTree

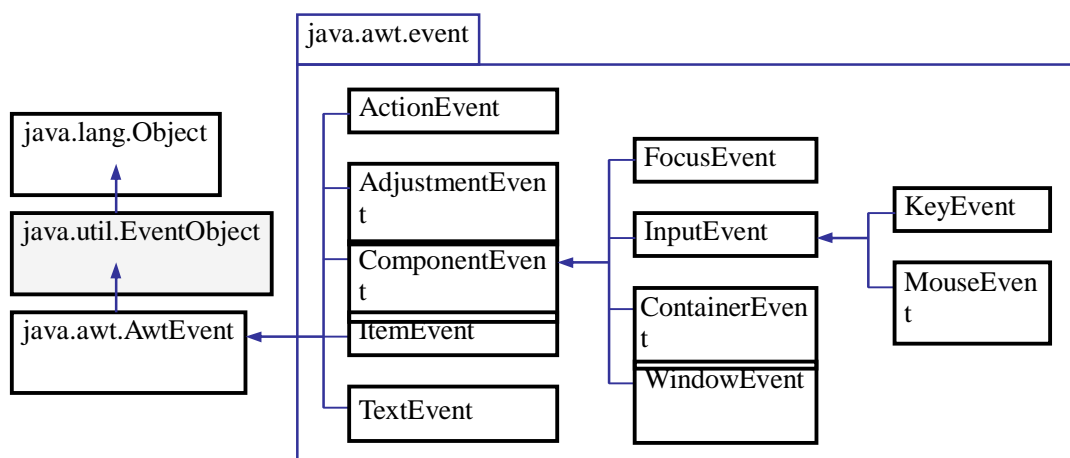
四：GUI 事件的处理

1: Java 事件和事件处理

你肯定已经注意到，以前的 Java GUI 应用程序不能正常地与用户进行交互，用户鼠标点击 Button 按钮时程序并无任何动作，这远不能满足我们的需要。如何让 Java GUI 动起来、对用户的操作做出实时处理，而不是作为静态画面来欣赏是本节学习的目标。

从 JDK1.1 开始，Java 采用了一种名为“事件授权模型”（Event Delegation Model）的事件处理机制，以支持 Java GUI 程序与用户的实时交互。与 Java 异常处理机制类似，事件处理机制的基本原理如下：

事先定义多种事件类型（见图 9-12），用以描述发生了什么事情（用户在 GUI 组件上进行的操作）。再约定各种 GUI 组件在与用户交互时，会触发相应的事件，即自动创建事件类对象并提交给 Java 运行时系统。系统接收到事件类对象后，立即将其发送给专门的对象，该对象调用其事件处理方法，处理事件对象，实现程序规定的逻辑功能。



这一模型涉及到的主要概念有：

事件（Event） ---- 一个对象，它描述了发生什么事情。

事件源（Event source） ----产生事件的组件对象，如按钮。

事件处理方法（Event handler） ----能够接收、解析和处理事件类对象、实现和用户交互的方法。

事件监听器（Event listener） ----调用事件处理方法的对象。

为使学习者更好地理解这一机制，举一个具有现实意义的例子：总统专机 A1（事件源）受到恐怖分子（用户）导弹攻击时，会自动发出警报 E1（事件），警报由飞行控制中心（运行时系统）接收后，立即转发给护航的战斗机 F1（事件监听器），F1 立即采取必要的防卫和反击措施（调用事件处理方法）。

示例 Java 事件处理

程序: TestActionEvent.java

```
import java.awt.*;
import java.awt.event.*;
public class TestActionEvent {
    public static void main(String args[]) {
        Frame f = new Frame("Test");
        Button b = new Button("Press Me!");
        Monitor bh = new Monitor();
        b.addActionListener(bh);
        f.add(b, BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
}

class Monitor implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("a button has been pressed");
    }
}
```

程序 GUI 如图所示



TestActionEvent.java

程序图形用户界面

其运行效果为: 用户每次鼠标单击“Press Me!”按钮时, 程序都会在控制台窗口中输出一行字符串“a button has been pressed”。

程序中, Button 对象 b 作为事件源组件, 调用其成员方法 `addActionListener()` 与监听器对象 bh 建立了监听和被监听的关系, 这一过程称为注册监听器。然后, 当按钮 b 受到用户鼠标单击时, 会触发 `ActionEvent` 事件, 即创建一个 `ActionEvent` 类型的对象并将之提交给运行时系统。运行时系统再将此其转发给此前曾在 b 上注册过的监听器 bh, 并以其做为实参自动调用 bh 的相应事件处理方法 `actionPerformed()`。

Java GUI 设计中, 就是通过这种注册监听器的方式对所关注的事件源进行监控的。如果关注某个组件产生的事件, 就可以在该组件上注册适当的监听器。那么到底有哪些常见的事件呢? 这里罗列了一些常见的事件:

`action(Event evt, Object what)`

当典型的事件针对组件发生 (例如, 当按下一个按钮或下拉列表项目被选中) 时调用。

`keyDown(Event evt, int key)`

当按键被按下, 组件拥有焦点时调用。第二个自变量是按下的键并且是冗余的是从 `evt.key` 处复制来的。

keyup(Event evt, int key)

当按键被释放, 组件拥有焦点时调用。

lostFocus(Event evt, Object what)

焦点从目标处移开时调用。通常, what 是从 evt.arg 里冗余复制的。

gotFocus(Event evt, Object what)

焦点移动到目标时调用。

mouseDown(Event evt, int x, int y)

一个鼠标按下存在于组件之上, 在 X, Y 坐标处时调用。

mouseUp(Event evt, int x, int y)

一个鼠标升起存在于组件之上时调用。

mouseMove(Event evt, int x, int y)

当鼠标在组件上移动时调用。

mouseDrag(Event evt, int x, int y)

鼠标在一次 mouseDown 事件发生后拖动。所有拖动事件都会报告给内部发生了 mouseDown

事件的那个组件, 直到遇到一次 mouseUp 为止。

mouseEnter(Event evt, int x, int y)

鼠标从前不在组件上方, 但目前在此。

mouseExit(Event evt, int x, int y)

鼠标曾经位于组件上方, 但目前不在。

接下来读者可能的困惑是, 什么样的对象才能做监听器? 什么样的方法才算是事件处理方法? 注册监听器的方法有那些?

首先, 并不是任何类型的对象都能做事件监听器, 相信没人会指望 String 对象能处理 ActionEvent 事件。JDK 的 java.awt.event 包中定义了一系列的事件监听器接口, 在这些接口中定义了各种 Java GUI 事件处理方法, 只有这些接口的实现类的对象才有资格做为监听器, 去处理相应的事件。事件监听器类型和对应的事件处理方法都是事先约定好的, 例如 ActionListener 接口中定义的 actionPerformed() 方法是专门用于处理 ActionEvent 类型事件的, 其格式如下:

```
public void actionPerformed(ActionEvent e);
```

其次, 注册监听器的方法是在 Java GUI 组件类中定义的, 其格式为:

```
public void addXXXListener(XXXListener l)
```

其中的XXX代表某种事件类型, 一个 Java GUI 组件可能产生多种不同类型的事件, 因而可以注册多种不同的监听器。例如, 当用户鼠标点击 Button 按钮时还会同时产生 MouseEvent 事件, Fram 窗口组件可以产生 MouseEvent 和 WindowEvent 等, 这是为了实现程序与用户的有效交互。在 java.awt.event 包中定义的相应的事件类型, 注册监听器时应指明该监听器所监控的事件种类。

例如示例中, Button 类的 addActionListener() 方法:

```
public void addActionListener(ActionListener l)
```


表 Java GUI 事件监听器接口

事件类型	相应监听器接口	监听器接口中的方法
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

2: 多重监听器

由于事件源可以产生多种不同类型的事件,因而可以注册(触发)多种不同类型的监听器。是当事件源发生了某种类型的事件时,只触发事先已就该种事件类型注册过的监听器。需要注意的两点是:

针对同一个事件源的同一种事件也可以注册多个监听器。如同为总统请多个保镖,他们都监听同一个事件源可能产生的同一种事件,即总统遇刺事件。

同一个监听器对象可以被同时注册到多个不同的事件源上。如同一支消防队伍同时负责多家工厂的安全,处理可能发生的火灾事件。

示例使用多重监听器

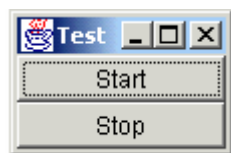
程序: TestActionEvent2.java

```
import java.awt.*;
```

```
import java.awt.event.*;
public class TestActionEvent2 {
    public static void main(String args[]) {
        Frame f = new Frame("Test");
        Button b1 = new Button("Start");
        Button b2 = new Button("Stop");
        Monitor2 bh = new Monitor2();
        b1.addActionListener(bh);
        b2.addActionListener(bh);
        b2.setActionCommand("game over");
        f.add(b1, "North");
        f.add(b2, "Center");
        f.pack();
        f.setVisible(true);
    }
}

class Monitor2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
    }
}
```

程序 GUI 如图所示



TestActionEvent2.java
程序图形用户界面

其运行效果为: 用户点击“Start”按钮时, 程序会在控制台窗口中输出一行字符串“Start”; 用户点击“Stop”按钮时, 程序输出“game over”。

程序 TestActionEvent2.java 中, 在 Button 组件 b1 和 b2 注册了同一个监听器对象 bh, 因此 b1 和 b2 被点击时产生的 ActionEvent 事件均被传送给 bh, 并由 bh 调用其约定的方法 actionPerformed() 进行处理。

需要说明的是, Button 类中定义了一个 String 类型属性 actionCommand, 用于记录该按钮对象所激发的 ActionEvent 事件的命令名称, 此属性的默认值为该按钮的标签字符串, 也可以使用 Button 类的成员方法 setActionCommand() 重置它的值。ActionEvent 类中也包含了一个同名属性 actionCommand, 其作用是: 当 ActionEvent 对象被创建时, 触发了此事件的事件源组件 (例如示例中的 Button 对象 b1 或 b2) 将把其自身的 actionCommand 属性值复制给新创建的 ActionEvent 对象的同名属性。随着 ActionEvent 对象的传递, 有关事件源组件的个性化信息作为事件对象的属性被发送到事件处理方法 actionPerformed() 中, 再通过 ActionEvent 类的 getActionCommand() 进行解析, 以实现事件源的区分和不同处理。

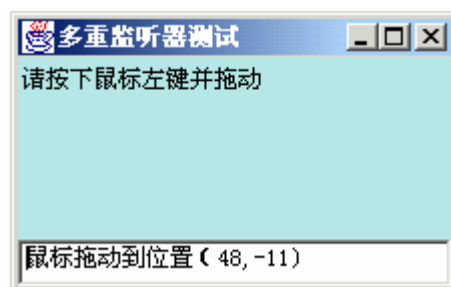
程序 TestActionEvent2.java 的运行结果可能每次都不相同,因为它受两个 Button 组件触发 ActionEvent 事件的时机和次序、即与用户交互情况的影响。这种程序运行方式称为事件驱动 (Event-driven)。

示例复杂多重监听器

程序: TestMultiListener.java

```
import java.awt.*;
import java.awt.event.*;
public class TestMultiListener implements
    MouseMotionListener,MouseListener {
    Frame f = new Frame("多重监听器测试");
    TextField tf = new TextField(30);
    public TestMultiListener(){
        Label l = new Label("请按下鼠标左键并拖动");
        f.add(l, "North");
        f.add(tf, "South");
        f.setBackground(new Color(180,225,225));
        f.addMouseMotionListener(this);
        f.addMouseListener(this);
        f.setSize(300, 200);
        f.setVisible(true);
    }
    public static void main(String args[]) {
        TestMultiListener t = new TestMultiListener();
    }
    public void mouseDragged(MouseEvent e) {
        String s = "鼠标拖动到位置 (" + e.getX() +
            ", " + e.getY() + ")";
        tf.setText(s);
    }
    public void mouseEntered(MouseEvent e) {
        tf.setText("鼠标已进入窗体");
    }
    public void mouseExited(MouseEvent e) {
        tf.setText("鼠标已移出窗体");
    }
    public void mouseMoved(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
}
```

程序 GUI 如图所示



程序 TestMultiListener.java GUI

其运行效果为: 用户鼠标移入(移出) Frame 窗口范围时, TextField (单行文本框) 组件中会显示“鼠标已进入(移出)窗体”的信息; 当用户鼠标在窗体范围内按下左键并拖动时, TextField 组件中会显示鼠标的当前位置坐标。

程序 TestMultiListener.java 中, 用到了 MouseMotionListener 和 MouseListener 两种常用的事件监听器接口。他们都是用来处理组件所触发的鼠标动作事件 MouseEvent 的, 但有不同的分工。

MouseListener 负责接收和处理鼠标的 press, release, click, enter 和 exit (即按下、松开、点击、移入、移出) 动作触发的 MouseEvent 事件, 其相应的事件处理方法为:

```
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseClicked(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
```

MouseMotionListener 负责接收和处理鼠标的 move, drag (即移动、拖动) 动作触发的 MouseEvent 事件, 其相应的事件处理方法为:

```
public void mouseMoved(MouseEvent e);
public void mouseDragged(MouseEvent e);
```

3: 事件适配器

为简化编程, 针对大多数事件监听器接口定义了相应的实现类----事件适配器类, 在适配器类中, 实现了相应监听器接口中所有的方法, 但不做任何事情。然后程序员在定义监听器类时就可以继承事件适配器类, 并只重写所需要的方法。

先来看一个事件适配器类 WindowAdapter 的定义:

```
public abstract class WindowAdapter implements WindowListener {
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
}
```

```
public void windowActivated(WindowEvent e) {}  
public void windowDeactivated(WindowEvent e) {}  
}
```

常用的事件适配器类有:

ComponentAdapter (组件事件适配器)

ContainerAdapter (容器事件适配器)

FocusAdapter (焦点事件适配器)

KeyAdapter (键盘事件适配器)

MouseAdapter (鼠标事件适配器)

MouseMotionAdapter (鼠标运动事件适配器)

WindowAdapter (窗口事件适配器)


适配器类只是为了简化编程而提供的一种中间性转换工具,使程序员在定义监听器类时可以不因直接实现监听器接口而被迫重写所有的抽象方法。但因 Java 单继承机制的限制,如果要定义的监听器类需要同时能处理两种以上的 GUI 事件,则只能直接实现有关的监听器接口,而无法只通过继承适配器实现。或者,一旦继承了适配器类,则无法再继承其他所需的父类。

示例事件监听器接口和适配器类使用对比

程序: TestListener.java

```
import java.awt.*;  
import java.awt.event.*;  
public class TestListener{  
    public static void main(String args[]){  
        Frame f = new Frame("Java Gui");  
        f.setSize(150,150);  
        MyListener m = new MyListener();  
        f.addWindowListener(m);  
        f.setVisible(true);  
    }  
}  
  
class MyListener implements WindowListener{  
    public void windowOpened(WindowEvent e){}  
    public void windowClosing(WindowEvent e){  
        System.exit(1);  
    }  
    public void windowClosed(WindowEvent e){}  
    public void windowIconified(WindowEvent e){}  
    public void windowDeiconified(WindowEvent e){}  
    public void windowActivated(WindowEvent e){}
```

```
public void windowDeactivated(WindowEvent e){}
}
```

示例程序运行时, 用户鼠标点击其 GUI 窗口角标可以关闭窗口并退出当前程序。其中, System 类的 exit() 方法可以终止当前正在运行的 Java 虚拟机。其格式为:

public static void exit(int status), 其中参数 status 为返回给操作系统的状态码, status 取值为 0 表示 JVM 正常结束, 非 0 表示非正常结束。

将程序 TestListener.java 中的 MyListener 类改为继承事件适配器类, 可以实现完全相同的功能, 但在编码上大大简化了。其形式如下:

```
class MyListener extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.exit(1);
    }
}
```

4: 内部类和匿名类在 Java 事件处理中的应用

在 JavaGUI 事件处理中, 常常采用内部类的形式来定义监听器类, 这是因为监听器类中封装的商务逻辑具有非常强的针对性, 通常没有重用价值。而且作为内部类的监听器类对象可以直接访问其封装类的成员, 这可以提供很大的便利。让我们来分析一个内部类做监听器的例子:

示例 在 GUI 事件处理中使用内部类

程序: TestInner.java

```
import java.awt.*;
import java.awt.event.*;
public class TestInner {
    Frame f = new Frame("内部类测试");
    TextField tf = new TextField(30);
    public TestInner(){
        f.add(new Label("请按下鼠标左键并拖动"), "North");
        f.add(tf, "South");
        f.setBackground(new Color(120, 175, 175));
        f.addMouseMotionListener(new InnerMonitor());
        f.addMouseListener(new InnerMonitor());
        f.setSize(300, 200);
        f.setVisible(true);
    }
    public static void main(String args[]) {
        TestInner t = new TestInner();
    }

    private class InnerMonitor
    implements MouseMotionListener, MouseListener {
        public void mouseDragged(MouseEvent e) {
```

```

        String s = "鼠标拖动到位置 (" + e.getX() + ", " + e.getY() + ")";
        tf.setText(s);
    }

    public void mouseEntered(MouseEvent e) {
        String s = "鼠标已进入窗体";
        tf.setText(s);
    }

    public void mouseExited(MouseEvent e) {
        String s = "鼠标已移出窗体";
        tf.setText(s);
    }

    public void mouseMoved(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
}
}

```

程序运行效果同程序 `TestMultiListener.java`, 其中用到的监听器类 `InnerMonitor` 被定义为内部类, 因此可以直接访问封装类的属性 `tf`, 以实现显示内容的改变。请读者思考如果把监听器类 `InnerMonitor` 改为一个普通的顶层类, 要实现相同的功能可能遇到的问题。

以内部类做为监听器类的方式还可能进一步简化, 这就是匿名内部类----在定义类的同时直接创建并一次性地使用该类对象, 而不指定类的名称。看先一个使用匿名类的例子:

示例 在 GUI 事件处理中使用匿名类

程序: `TestAnonymous.java`

```

import java.awt.*;
import java.awt.event.*;
public class TestAnonymous {
    Frame f = new Frame("匿名内部类测试");
    TextField tf = new TextField(30);
    public TestAnonymous(){
        f.add(new Label("请按下鼠标左键并拖动"), "North");
        f.add(tf, "South");
        f.addMouseListener(
new MouseMotionAdapter(){
            public void mouseDragged(MouseEvent e) {
                tf.setText("鼠标位置" + e.getPoint());
            }
        });
        f.setSize(300, 200);
        f.setVisible(true);
    }
}

```

```
public static void main(String args[]) {  
    TestAnonymous t = new TestAnonymous();  
}  
}
```

程序运行效果与示例程序 `TestMultiListener.java` 基本相同, 查看其中监听器对象的创建方式, 会有一些奇怪的发现: `new MouseMotionAdapter(){...}`, 看来是要创建一个抽象类 `MouseMotionAdapter` 类型的对象, 这似乎是不可能的。而且, 在 `new MouseMotionAdapter()` 后面紧跟的大括号又是什么意思? 这怎么可能是合法的 Java 代码? 实际上, 这些代码是合法的, 它的含义是: 定义 `MouseMotionAdapter` 的一个匿名子类, 并创建一个该子类的对象。上面提及的花括号内容就是子类的类体, 子类中重写了父类的 `mouseDrag()` 方法。

注意: 在匿名类中不能声明构造方法, 因为构造方法的名字要和所处的类名一样, 而匿名是没有名字的。

匿名内部类在本质上是普通内部类的一种简化形式, 可以进行相互转换。例如, 下面两段程序完全等价:

示例 内部类和匿名内部类使用比较

程序 1: `Test1.java`

```
import java.awt.*;  
import java.awt.event.*;  
  
public class Test1{  
    Frame f = new Frame("Java Gui");  
    public Test1(){  
        MyInner m = new MyInner();  
        f.addWindowListener(m);  
        f.setSize(150, 150);  
        f.setVisible(true);  
    }  
    public static void main(String args[]){  
        new Test1();  
    }  
    class MyInner extends WindowAdapter{  
        public void windowClosing(WindowEvent e){  
            System.exit(1);  
        }  
    }  
}
```

程序 2: `Test2.java`

```
import java.awt.*;  
import java.awt.event.*;  
public class Test2{  
    Frame f = new Frame("Java Gui");  
    public Test2(){  
        f.addWindowListener(new WindowAdapter(){  

```

```
        public void windowClosing(WindowEvent e){
            System.exit(1);
        }
    });
    f.setSize(150,150);
    f.setVisible(true);
}
public static void main(String args[]){
    new Test2();
}
}
```

匿名类也可以从接口中派生，其格式为 `new <interface_name>(){...}`。此时，匿名类必须实现接口中所有的抽象方法。除此之外，匿名类具备内部类的全部特性，比如可以根据需要添加新的属性和方法，或访问其封装类的成员。

示例 使用匿名内部类实现接口

程序 1: TestAnonymous2.java

```
import java.awt.*;
import java.awt.event.*;
public class TestAnonymous2 {
    Frame f = new Frame("Test");
    TextField tf = new TextField(10);
    Button b1 = new Button("Start");
    public TestAnonymous2(){
        f.add(b1,"North");
        f.add(tf,"South");
        b1.addActionListener(new ActionListener(){
            private int i;
            public void actionPerformed(ActionEvent e) {
                tf.setText(e.getActionCommand() + ++i);
            }
        });
        f.pack();
        f.setVisible(true);
    }
    public static void main(String args[]) {
        new TestAnonymous2();
    }
}
```

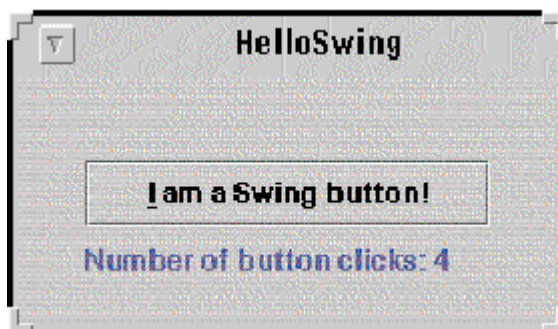
程序中的匿名内部类实现了 `ActionListener` 接口，其中添加了新的属性 `i`，用做计数器。需要注意的是，匿名内部类无法同时实现多个接口。

五: 基本的 Swing 程序

1: 一个基本的示例程序

下面详细讲述一个 Swing 的应用程序 HelloSwing。

HelloSwing 应用程序的输出产生下图所示的窗口:



每次用户点击按钮时, 标签就会更新。

1.1: HelloSwing

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import java.awt.accessibility.*;
5.
6. public class HelloSwing implements ActionListener {
7.     private JFrame jFrame;
8.     private JLabel jLabel;
9.     private JPanel jPanel;
10.    private JButton jButton;
11.    private AccessibleContext accContext;
12.
13.    private String labelPrefix =
14.    "Number of button clicks: ";
15.    private int numClicks = 0;
16.
17.    public void go() {
18.
19.    // Here is how you can set up a particular
20.    // LookAndFeel. Not necessary for default.
21.    //
22.    // try {
23.    // UIManager.setLookAndFeel (
24.    // UIManager.getLookAndFeel());
```



```
25.// } catch (UnsupportedLookAndFeelException e) {
26.// System.err.println("Couldn't use the " +
27.// "default look and feel " + e);
28.// }
29.
30.jFrame = new JFrame("HelloSwing");
31.jLabel = new JLabel (labelPrefix + "0");
32.
33.jButton = new JButton("I am a Swing button!");
34.
35.// Create a shortcut: make ALT-A be equivalent
36.// to pressing mouse over button.
37.jButton.setMnemonic('i');
38.
39.jButton.addActionListener(this);
40.
41.// Add support for accessibility.
42.accContext = jButton.getAccessibleContext();
43.accContext.setAccessibleDescription(
44."Pressing this button increments " +
45."the number of button clicks");
46.
47.// Set up pane.
48.// Give it a border around the edges.
49.jPanel = new JPanel ();
50.jPanel.setBorder(
51.BorderFactory.createEmptyBorder(
52.30,30,10,30));
53.
54.// Arrange for compts to be in a single column.
55.jPanel.setLayout(new GridLayout(0, 1));
56.
57.// Put compts in pane, not in JFrame directly.
58.jPanel.add(jButton);
59.jPanel.add(jLabel);
60.jFrame.setContentPane(jPanel);
61.
62.// Set up a WindowListener inner class to handle
63.// window's quit button.
64.WindowListener wl = new WindowAdapter() {
65.public void windowClosing(WindowEvent e) {
66.System.exit(0);
67.}
68.};
```

```
69. JFrame.addWindowListener(wl);
70.
71. JFrame.pack();
72. JFrame.setVisible(true);
73. }
74.
75. // Button handling.
76. public void actionPerformed(ActionEvent e) {
77. numClicks++;
78. jLabel.setText(labelPrefix + numClicks);
79. }
80.
81. public static void main(String[] args) {
82.
83. HelloSwing helloSwing = new HelloSwing();
84. helloSwing.go();
85. }
86. }
```

1.2: 导入 Swing 包

- | 导入 Swing 包
- | 选择外观和感觉
 - | getLookAndFeel()
- | 设置窗口容器
 - | JFrame 与 Frame 相似
 - | 你不能直接将组件加入到 JFrame 中
 - | 一个 content pane 包含了除菜单条外所有 Frame 的可视组件

语句行 `import javax.swing.*` 装入整个 Swing 包, 它包括了标准 Swing 组件和功能。
选择外观和感觉

HelloSwing 的第 22—28 行给定了应用程序外观和感觉的格式。`getLookAndFeel()` 方法返回在 Windows 环境中的外观和感觉。在运行 Solaris 操作系统的机器上, 这个方法则返回一个公共桌面环境 (CDE) /Motif 的外观和感觉。因为都是缺省值, 所以对本例来说, 这些行都不是必需的。

1.3: 建立窗口

Swing 程序用 JFrame 对象实现了它们的窗口。JFrame 类是 AWT Frame 类的一个子类。它还加入了一些 Swing 所独有的特性。HelloSwing 中, 处理 JFrame 的代码如下:

```
public HelloSwing() {
    JFrame jFrame;
```

```

JPanel jPanel;
.....
JFrame = new JFrame("HelloSwing");
    jPanel = new JPanel();
    .....
    JFrame.setContentPane(jPanel);

```

这段代码与使用 Frame 的代码十分相似。唯一的区别在于,你不能将组件加入到 JFrame 中。你可以或者将组件加入到 JFrame 的 content pane 中,或者提供一个新的 content pane。

一个 content pane 是一个包含除菜单条(如果有的话)外所有框架的可视组件的容器。要获得一个 JFrame 的 content pane,可使用 getContentPane()方法。要设置它的 content pane(如前面本例所示),则可使用 setContentPane()方法。

1.4: 建立 Swing 组件

Swing 应用程序基础	
I	建立 Swing 组件
I	Hello Swing.java 示例实例化了 4 个 Swing 组件,这 4 个组件是: JFrame, JButton, JLabel 和 JPanel
I	支持辅助技术
I	Hello Swing.java 示例代码支持辅助技术 <pre> accContext = jButton.getAccessibleContext(); accContext.setAccessibleDescription("Pressing this button increments " + " the number of button clicks."); </pre>

Hello Swing 程序显式地实例化了 4 个组件: JFrame, JButton, JLabel 和 JPanel。Hello Swing 用第 33—45 行中的代码来初始化 JButton。

第 33 行创建了按钮。第 37 行将 ACT—I 键组合设置为快捷键,用来模拟按钮的点击。第 39 行为点击注册了一个事件处理器。第 41—45 行描述了一个按钮,使得辅助技术可以提供有关按钮功能的信息。

第 49—59 行初始化了 JPanel。这些代码创建了 JPanel 对象,设置它的边框,并将它的布局管理器设置为单列地放置 panel 的内容。最后,将一个按钮和一个标签加入到 Panel 中。Hello Swing 中的 Panel 使用了一个不可见的边框,用来在它周围放入额外的填充。

1.5: 支持辅助技术

Hello Swing.java 中唯一支持辅助技术的代码是:

```

accContext = jButton.getAccessibleContext();
accContext.setAccessibleDescription(
    "Pressing this button increments " +
    " the number of button clicks.");

```

下列信息集也可由辅助技术使用:

```

jButton = new JButton("I'm a Swing button!");
jLabel = new JLabel(labelPrefix + "0");
jLabel.setText(labelPrefix + numClicks);

```

在 JFrame, JButton, JLabel 和其他所有组件中, 都有内建的 Accessibility 支持。辅助技术可以很容易地获得文本, 甚至与一组件某特定部分相关的文本。

2: 构造一个 Swing GUI

Swing 包定义了两种类型的组件:

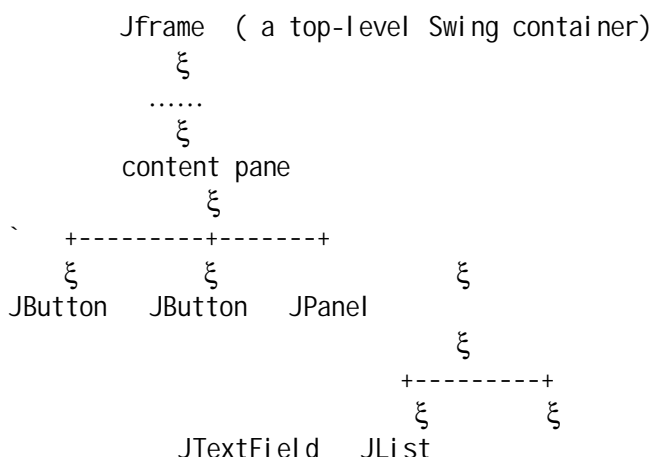
- I 顶层容器 (JFrame, JApplet, JWindow, 和 JDialog)
- I 轻质组件 (其他的 J..., 如 JButton, JPanel 和 JMenu)

顶层容器定义了可以包含轻质组件的框架。特别地, 一个顶层 Swing 容器提供了一个区域, 轻质组件可在这个区域中绘制自身。顶层容器是它们对应的重质 AWT 组件的 Swing 子类。这些 Swing 容器依靠它们的 AWT 超类的本地方法与硬件进行适当的交互。

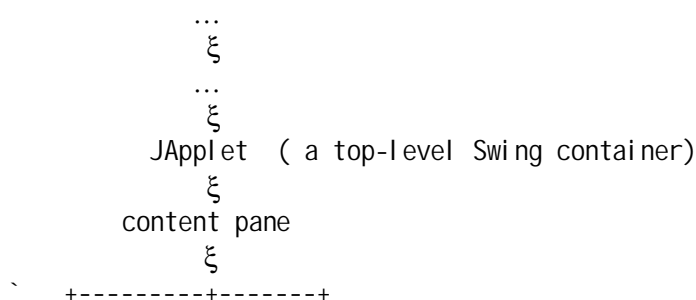
通常, 每个 Swing 组件在其容器层次结构中都应当有一个位于组件上面的顶层 Swing 容器。例如, 每个包含 Swing 组件的 Applet 都应作为 JApplet (而它自身又是 java.applet.Applet 的一个子类) 的子类来实现。相似地, 每个包含 Swing 组件的主窗口都应用 JFrame 来实现。典型地, 如果你在使用 Swing 组件, 你将只能使用 Swing 组件和 Swing 容器。

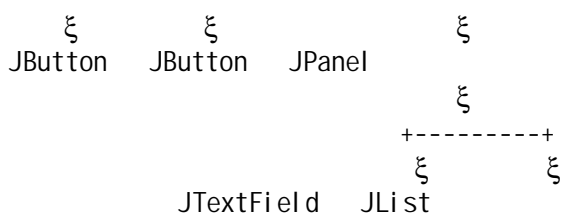
Swing 组件可以加入到一个与顶层容器关联的 content pane 中, 但绝不能直接加入到顶层容器中。content pane 是一个轻质 Swing 组件, 如 JPanel。

下面是一个典型 Swing 程序的 GUI 容器层次结构图, 这个程序实现了一个包含 2 个按钮, 一个文本域和一个列表:



下面是关于同样的 GUI 的另一个容器层次结构, 只是在这里, GUI 是在浏览器中运行的一个 Applet。





下面是构造如上图所示的 GUI 层次结构的代码:

```

1. import javax.swing.*;
2. import java.awt.*;
3.
4. public class SwingGUI {
5.
6.     JFrame topLevel;
7.     JPanel jPanel;
8.     JTextField jTextField;
9.     JList jList;
10.
11.     JButton b1;
12.     JButton b2;
13.     Container contentPane;
14.
15.     Object listData[] = {
16.         new String("First selection"),
17.         new String("Second selection"),
18.         new String("Third selection")
19.     };
20.
21.     public static void main (String args[]) {
22.         SwingGUI swingGUI = new SwingGUI ();
23.         swingGUI.go();
24.     }
25.
26.     public void go() {
27.         topLevel = new JFrame("Swing GUI");
28.
29.         // Set up the JPanel, which contains the text field
30.         // and list.
31.         jPanel = new JPanel ();
32.         jTextField = new JTextField(20);
33.         jList = new JList(listData);
34.
35.         contentPane = topLevel.getContentPane();
36.         contentPane.setLayout(new BorderLayout());
37.

```

```
38. b1 = new JButton("1");
39. b2 = new JButton("2");
40. contentPane.add(b1, BorderLayout.NORTH);
41. contentPane.add(b2, BorderLayout.SOUTH);
42.
43. jPanel1.setLayout(new FlowLayout());
44. jPanel1.add(jTextField1);
45. jPanel1.add(jList1);
46. contentPane.add(jPanel1, BorderLayout.CENTER);
47.
48. topLevel.pack();
49. topLevel.setVisible(true);
50. }
51. }
```

六：谈谈 GUI 编程中的“写”界面

前面已经总结了如何“画”界面，画出来的界面是静态的，如何让它“动”起来呢？下面就谈谈如何在画好的界面上写代码，通过代码来实现动态的功能。

1：表现层三大功能

应用程序的界面层也被称为表现层，专门用于图形化的跟用户进行交互。通常表现层具有如下 3 个主要的功能：

- (1)：展示数据：主要是从逻辑层获取需要展示给用户看或操作的数据
- (2)：人机交互：用户可以在界面上输入值，也可以点击某些组件，从而引起某些动态的事件
- (3)：收集界面参数，调用逻辑层接口

2：如何写代码呢

分析表现层的功能，需要写代码来完成的功能不多，主要有两个部分，其一是展示数据部分，其二是事件处理。写代码的通常步骤总结如下：

展示数据部分的代码通常是写在界面初始化的方法里面，写法如下：

- (1)：调用逻辑层接口，获取需要展示的数据
- (2)：转换数据
- (3)：把数据设置到组件上进行展示

事件处理部分的代码通常是写在事件处理的方法里面，写法如下：

- (1)：收集参数
- (2)：组织参数
- (3)：调用逻辑层接口，获取返回值
- (4)：根据返回值，选择下一个界面

练习实践课:

- I Swing 基础
- I 事件处理

程序 1

鼠标移动

需求:

- 1、不论鼠标何时移动都围绕它画一个小圆;
- 2、每当按下鼠标时, 屏幕显示文字。

目标:

- 1、鼠标移动事件;
- 2、绘图语句;
- 3、色彩处理。

程序:

```
//: BangBean.java
package com.useful.java.part4;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class BangBean extends Canvas implements Serializable {
    protected int xm, ym;
    protected int cSize = 20; // Circle size
    protected String text = "Bang!";
    protected int fontSize = 48;
    protected Color tColor = Color.red;
    protected ActionListener actionListener;
    public BangBean() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }
    public int getCircleSize() { return cSize; }
    public void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public String getBangText() { return text; }
    public void setBangText(String newText) {
        text = newText;
    }
}
```

```
}
public int getFontSize() { return fontSize; }
public void setFontSize(int newSize) {
    fontSize = newSize;
}
public Color getTextColor() { return tColor; }
public void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
              cSize, cSize);
}
// This is a unicast listener, which is
// the simplest form of listener management:
public void addActionListener (
    ActionListener l)
    throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = l;
}
public void removeActionListener(
    ActionListener l) {
    actionListener = null;
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font(
                "TimesRoman", Font.BOLD, fontSize));
        int width =
            g.getFontMetrics().stringWidth(text);
        g.drawString(text,
            (getSize().width - width) /2,
            getSize().height/2);
        g.dispose();
        // Call the listener's method:
        if(actionListener != null)
            actionListener.actionPerformed(
                new ActionEvent(BangBean.this,
```



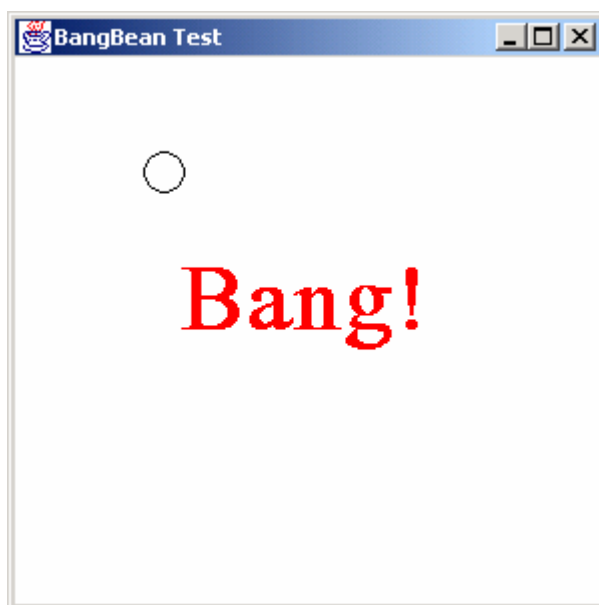
```
        ActionEvent.ACTION_PERFORMED, null));
    }
}
class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}
public Dimension getPreferredSize() {
    return new Dimension(200, 200);
}
// Testing the BangBean:
public static void main(String[] args) {
    BangBean bb = new BangBean();
    try {
        bb.addActionListener(new BBL());
    } catch (TooManyListenersException e) {}
    Frame aFrame = new Frame("BangBean Test");
    aFrame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    aFrame.add(bb, BorderLayout.CENTER);
    aFrame.setSize(300,300);
    aFrame.setVisible(true);
}
// During testing, send action information
// to the console:
static class BBL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("BangBean action");
    }
}
}
```

说明:

- 1、你可以尝试改变一下此程序，如圆的特征，色彩等；
- 2、BangBean 同样拥有它自己的addActionListener()和removeActionListener()方法，因此我们可以附上自己的当用户单击在 BangBean 上时会被激活的接收器。这

样,我们将能够确认可支持的属性和事件,最重要的是我们会注意到 **BangBean** 执行了这种串联化的接口。这意味着应用程序构建工具可以在程序设计者调整完属性值后利用串联为 **BangBean** 贮藏所有的信息。当 **Bean** 作为运行的应用程序的一部分被创建时,那些被贮藏的属性被重新恢复,因此我们可以正确地得到我们的设计。

3、程序运行如下:



作业

1: 完成地址本的应用,为已经画出的增、删、改、查、列表界面添加动态功能的实现。要求数据存储使用第九章集合完成的作业。也就是使用 **Swing** 做表现层,集合做逻辑层来实现可以增删改查的地址本的功能。

第十二章 输入/输出流

教学目标:

i I/O 包简介

i 掌握文件和过滤器□

i 掌握 Reader 和 Writer

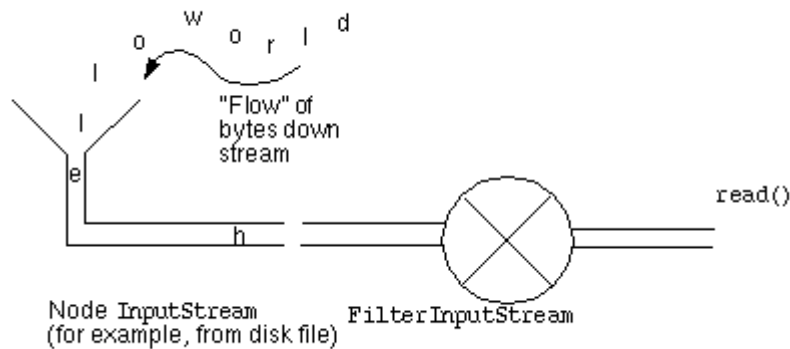
i 掌握二进制流的读写

i 掌握文本流的读写

一: Java 的 io 包

1: 流是什么

简单的说: 流是字节从源到目的地运行的轨迹。
次序是有意义的, 字节会按照次序进行传递。



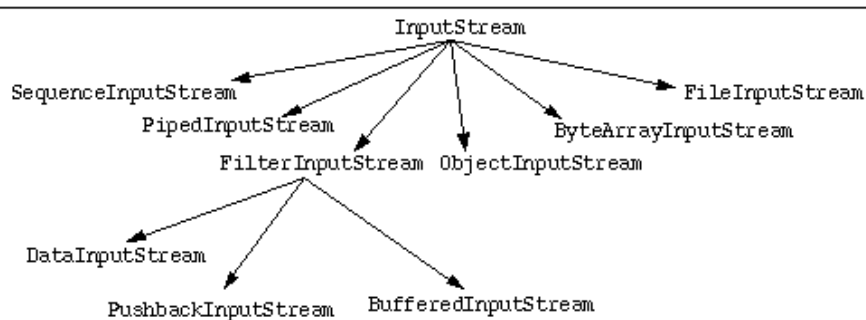
2: 流有什么

两种基本的流是: 输入流和输出流。你可以从输入流读, 但你不能对它写。要从输入流读取字节, 必须有一个与这个流相关联的字符源。

在 java.io 包中, 有一些流是结点流, 即它们可以从一个特定的地方读写, 例如磁盘或者一块内存。其他流称作过滤器。一个过滤器输入流是用一个到已存在的输入流的连接创建的。此后, 当你试图从过滤输入流对象读时, 它向你提供来自另一个输入流对象的字符。

2.1: IO 包中基本流类

在 java.io 包中定义了一些流类。下图表明了包中的类层次。一些更公共的类将在后面介绍。



2.1.1: FileInputStream 和 FileOutputStream

这些类是结点流, 而且正如这个名字所暗示的那样, 它们使用磁盘文件。这些类的构造函数允许你指定它们所连接的文件。要构造一个 `FileInputStream`, 所关联的文件必须存在而且是可读的。如果你要构造一个 `FileOutputStream` 而输出文件已经存在, 则它将被覆盖。

`FileInputStream infile =`

```
new FileInputStream("myfile.dat");
FileOutputStream outfile =
    new FileOutputStream("results.dat");
```

2.1.2 ByteArrayInputStream 和 BufferedOutputStream

这些是过滤器流，它们可以提高 I/O 操作的效率。

2.1.3 DataInputStream 和 DataOutputStream

DataInputStream 方法

```
byte readByte()
long readLong()
double readDouble()
```

DataOutputStream 方法

```
void writeByte(byte)
void writeLong(long)
void writeDouble(double)
```

注意 DataInputStream 和 DataOutputStream 的方法是成对的。

这些流都有读写字符串的方法，但不应当使用这些方法。它们已经被后面所讨论的读者和作者所取代。

2.1.4 PipedInputStream 和 PipedOutputStream

管道流用来在线程间进行通信。一个线程的 PipedInputStream 对象从另一个线程的 PipedOutputStream 对象读取输入。要使管道流有用，必须有一个输入方和一个输出方。

3: InputStream 与 OutputStream 流能干什么

3.1: InputStream

InputStream 的作用是标志那些从不同起源地产生输入的类型。这些起源地包括（每个都有一个相关的 InputStream 子类）：

- (1) 字节数组
- (2) String 对象
- (3) 文件
- (4) “管道”，它的工作原理与现实生活中的管道类似：将一些东西置入一端，它们在另一端出来。
- (5) 一系列其他流，以便我们将其统一收集到单独一个流内。
- (6) 其他起源地，如 Internet 连接等（将在本书后面的部分讲述）。

除此以外，FilterInputStream 也属于 InputStream 的一种类型，用它可为“破坏器”类提供一个基础类，以便将属性或者有用的接口同输入流连接到一起。这将在以后讨论。

ByteArrayInputStream 允许内存中的一个缓冲区作为 InputStream 使用 从中提取字

节的缓冲区作为一个数据源使用。通过将其同一个 `FilterInputStream` 对象连接, 可提供一个有用的接口。

`StringBufferInputStream` 将一个 `String` 转换成 `InputStream` 一个 `String` (字符串)。基础的实施方案实际采用一个 `StringBuffer` (字符串缓冲) 作为一个数据源使用。通过将其同一个 `FilterInputStream` 对象连接, 可提供一个有用的接口。

`FileInputStream` 用于从文件读取信息 代表文件名的一个 `String`, 或者一个 `File` 或 `FileDescriptor` 对象作为一个数据源使用。通过将其同一个 `FilterInputStream` 对象连接, 可提供一个有用的接口。

`PipedInputStream` 产生为相关的 `PipedOutputStream` 写的的数据。实现了“管道化”的概念。 `PipedOutputStream` 作为一个数据源使用。通过将其同一个 `FilterInputStream` 对象连接, 可提供一个有用的接口。

`SequenceInputStream` 将两个或更多的 `InputStream` 对象转换成单个 `InputStream` 使用两个 `InputStream` 对象或者一个 `Enumeration`, 用于 `InputStream` 对象的一个容器 / 作为一个数据源使用。通过将其同一个 `FilterInputStream` 对象连接, 可提供一个有用的接口

`FilterInputStream` 对作为破坏器接口使用的类进行抽象; 那个破坏器为其他 `InputStream` 类提供了有用的功能。

3.1.1: InputStream 方法

三个 read 方法

```
int read()
int read(byte [])
int read(byte[], int ,int )
```

这三个方法提供对输入管道数据的存取。简单读方法返回一个 `int` 值, 它包含从流里读出的一个字节或者 -1, 其中后者表明文件结束。其它两种方法将数据读入到字节数组中, 并返回所读的字节数。第三个方法中的两个 `int` 参数指定了所要填入的数组的子范围。

注—考虑到效率, 总是在实际最大的块中读取数据。

```
void close()
```

你完成流操作之后, 就关闭这个流。如果你有一个流所组成的栈, 使用过滤器流, 就关闭栈顶部的流。这个关闭操作会关闭其余的流。

```
int available()
```

这个方法报告立刻可以从流中读取的字节数。在这个调用之后的实际读操作可能返回更多的字节数。

```
skip(long)
```

这个方法丢弃了流中指定数目的字符。

```
boolean markSupported()
void mark(int)
void reset()
```

如果流支持“回放”操作, 则这些方法可以用来完成这个操作。如果 `mark()` 和 `reset()` 方法可以在特定的流上操作, 则 `markSupported()` 方法将返回 `true`。 `mark(int)` 方法用

来指明应当标记流的当前点和分配一个足够大的缓冲区,它最少可以容纳参数所指定数量的字符。在随后的 `read()` 操作完成之后,调用 `reset()` 方法来返回你标记的输入点。

3.2 OutputStream

这一类别包括的类决定了我们的输入往何处去:一个字节数组(但没有 `String`;假定我们可用字节数组创建一个);一个文件;或者一个“管道”。

除此以外, `FilterOutputStream` 为“破坏器”类提供了一个基础类,它将属性或者有用的接口同输出流连接起来。这将在以后讨论。

`ByteArrayOutputStream` 在内存中创建一个缓冲区。我们发送给流的所有数据都会置入这个缓冲区。可选缓冲区的初始大小用于指出数据的目的地。若将其同 `FilterOutputStream` 对象连接到一起,可提供一个有用的接口。

`FileOutputStream` 将信息发给一个文件 用一个 `String` 代表文件名,或选用一个 `File` 或 `FileDescriptor` 对象用于指出数据的目的地。若将其同 `FilterOutputStream` 对象连接到一起,可提供一个有用的接口。

`PipedOutputStream` 我们写给它的任何信息都会自动成为相关的 `PipedInputStream` 的输出。实现了“管道化”的概念 `PipedInputStream` 为多线程处理指出自己数据的目的地将其同 `FilterOutputStream` 对象连接到一起,便可提供一个有用的接口。

`FilterOutputStream` 对作为破坏器接口使用的类进行抽象处理;那个破坏器为其他 `OutputStream` 类提供了有用的功能。

3.2.1: OutputStream 方法

三个基本的 `write()` 方法

- | `void write(int)`
- | `void write(byte [])`
- | `void write(byte [], int, int)`

这些方法写输出流。和输入一样,总是尝试以实际最大的块进行写操作。

void close()

当你完成写操作后,就关闭输出流。如果你有一个流所组成的栈,就关闭栈顶部的流。这个关闭操作会关闭其余的流。

void flush()

有时一个输出流在积累了若干次之后才进行真正的写操作。`flush()` 方法允许你强制执行写操作。

4: URL 输入流

除了基本的文件访问之外,Java 技术提供了使用统一资源定位器(URL)来访问网络上的文件。当你使用 `Applet` 的 `getDocumentBase()` 方法来访问声音和图象时,你已经隐含地使用了 URL 对象。

```
String imageFile = new String ("images/Duke/T1.gif");
images[0] = getImage(getDocumentBase(), imageFile);
```

然而,你必须象下面的程序那样提供一个直接的 URL

```
java.net.URL imageSource;
try {
    imageSource = new URL("http://mysite.com/~info");
} catch ( MalformedURLException e) {}
images[0] = getImage(imageSource, "Duke/T1.gif");
```

4.1 打开一个输入流

你可以通过存储文档基目录下的一个数据文件来打开一个合适的 URL 输入流。

```
1.InputStream is = null;
2.String datafile = new String("Data/data.1-96");
3.byte buffer[] = new byte[24];
4.try {
5.// new URL throws a MalformedURLException
6.// URL.openStream() throws an IOException
7.is = (new URL(getDocumentBase(),
    datafile)).openStream();
8.} catch (Exception e) {}
```

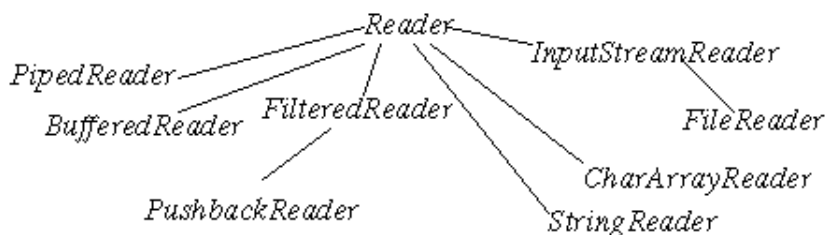
现在, 你可以就象使用 `FileInputStream` 对象那样来用 `it` 来读取信息:

```
1.try {
2.is.read(buffer, 0, buffer.length);
3.} catch (IOException e1) {}
```

警告—记住大多数用户进行了浏览器的安全设置, 以防止 Applet 存取文件。

5: Reader 和 Writer

Java 中的 Reader 和 Writer 的结构如下图



5.1 Uni code

Java 技术使用 Uni code 来表示字符串和字符, 而且它提供了 16 位版本的流, 以使用类似的方法来处理字符。这些 16 位版本的流称为读者和作者。和流一样, 它们都在 `java.io` 包中。

读者和作者中最重要的版本是 `InputStreamReader` 和 `OutputStreamWriter`。这些类用来作为字节流与读者和作者之间的接口。

当你构造一个 `InputStreamReader` 或 `OutputStreamWriter` 时, 转换规则定义了 16 位

Uni code 和其它平台的特定表示之间的转换。

缺省情况下,如果你构造了一个连接到流的读者和作者,那么转换规则会在缺省平台所定义的字节编码和 Uni code 之间切换。在英语国家中,所使用的字节编码是:ISO 8859-1。

你可以使用所支持的另一种编码形式来指定其它的字节编码。在 native2ascii 工具中,你可以找到一个关于所支持的编码形式的列表。

使用转换模式,Java 技术能够获得本地平台字符集的全部灵活性,同时由于内部使用 Uni code,所以还能保持平台独立性。

5.2 缓冲读者和作者

因为在各种格式之间进行转换和其它 I/O 操作很类似,所以在处理大块数据时效率最高。在 InputStreamReader 和 OutputStreamWriter 的结尾链接一个 BufferedReader 和 BufferedWriter 是一个好主意。记住对 BufferedWriter 使用 flush() 方法。

5.3 读入字符串输入

下面这个例子说明了从控制台标准输入读取字符串所应当使用的一个技术。

```
1.import java.io.*;
2.public class CharInput {
3.public static void main (String args[]) throws
4.java.io.IOException {
5.String s;
6.InputStreamReader ir;
7.BufferedReader in;
8.ir = new InputStreamReader(System.in);
9.in = new BufferedReader(ir);
10.
11.while ((s = in.readLine()) != null) {
12.System.out.println("Read: " + s);
13.}
14.}
15.}
```

5.4 使用其它字符转换

如果你需要从一个非本地(例如,从连接到一个不同类型的机器的网络连接读取)的字符编码读取输入,你可以象下面这个程序那样,使用显式的字符编码构造

```
ir=new InputStreamReader(System.in, "iso8859-1");
```

对于已有的字符串,还可以这样进行编码转换:

```
String s = new String(str.getBytes("ISO8859-1"), "GB2312");
```

注意:

(1): 如果你通过网络连接读取字符,就应该使用这种形式。否则,你的程序会总是试图将所读取的字符当作本地表示来进行转换,而这并不总是正确的。ISO 8859-1 是映射到 ASCII 的 Latin-1 编码模式。

(2): 用于表示中文的编码方式,常用的是“GB2312”或者“GBK”

6: 串行化（序列化）

将一个对象存放到某种类型的永久存储器上称为保持。如果一个对象可以被存放到磁盘或磁带上，或者可以发送到另外一台机器并存放到存储器或磁盘上，那么这个对象就被称为可保持的。

`java.io.Serializable` 接口没有任何方法，它只作为一个“标记者”，用来表明实现了这个接口的类可以考虑串行化。类中没有实现 `Serializable` 的对象不能保存或恢复它们的状态。

6.1 对象图

当一个对象被串行化时，只有对象的数据被保存；方法和构造函数不属于串行化流。如果一个数据变量是一个对象，那么这个对象的数据成员也会被串行化。树或者对象数据的结构，包括这些子对象，构成了对象图。

因为有些对象类所表示的数据在不断地改变，所以它们不会被串行化；例如，`java.io.FileInputStream`、`java.io.FileOutputStream` 和 `java.lang.Thread` 等流。如果一个可串行化对象包含对某个不可串行化元素的引用，那么整个串行化操作就会失败，而且会抛出一个 `NotSerializableException`。

如果对象图包含一个不可串行化的引用，只要这个引用已经用 `transient` 关键字进行了标记，那么对象仍然可以被串行化。

```
public class MyClass implements Serializable {  
    public transient Thread myThread;  
    private String customerID;  
    private int total;
```

域存取修饰符对于被串行化的对象没有任何作用。写入到流的数据是字节格式，而且字符串被表示为 UTF(文件系统安全的通用字符集转换格式)。`transient` 关键字防止对象被串行化。

```
public class MyClass implements Serializable {  
    public transient Thread myThread;  
    private transient String customerID;  
  
    private int total;
```

7: 对象流的基本读写示例

7.1 写

对一个文件流读写对象是一个简单的过程。考虑如下代码段，它将一个 `java.util.Date` 对象的实例发送到一个文件：

```
1. public class SerializeDate {  
2.     SerializeDate() {  
3.         Date d = new Date ();  
4.         try {  
5.             FileOutputStream f = new  
6.                 FileOutputStream("date.ser");  
7.             ObjectOutputStream s = new  
8.                 ObjectOutputStream(f);  
9.             s.writeObject (d);
```

```
10.f.close ();
11.} catch (IOException e) {
12.e.printStackTrace ();
13.}
14.}
15.
16.public static void main (String args[]) {
17.new SerializeDate();
18.}
19.}
```

7.2 读

读对象和写对象一样简单, 只需要说明一点—readObject()方法将流作为一个 Object 类型返回, 而且在使用那个类的方法之前, 必须把它转换成合适的类名。

```
1.public class UnSerializeDate {
2.UnSerializeDate () {
3.Date d = null;
4.try {
5.FileInputStream f = new
6.FileInputStream("date.ser");
7.ObjectInputStream s = new
8.ObjectInputStream(f);
9.d = (Date) s.readObject ();
10.f.close ();
11.} catch (Exception e) {
12.e.printStackTrace ();
13.}
14.
15.System.out.println("Unserialized Date object from date.ser");
16.System.out.println("Date: "+d);
17.}
18.
19.public static void main (String args[]) {
20.new UnSerializeDate();
21.}
22.}
```

二: 文件和过滤器

1: File 操作

File 类是 IO 中文件操作最常用的类。

File 类有一个欺骗性的名字——通常会认为它对付的是一个文件, 但实情并非如此。它

既代表一个特定文件的名称，也代表目录内一系列文件的名称。若代表一个文件集，便可用 `list()` 方法查询这个集，返回的是一个字符串数组。之所以要返回一个数组，而非某个灵活的集合类，是因为元素的数量是固定的。而且若想得到一个不同的目录列表，只需创建一个不同的 `File` 对象即可。事实上，“`FilePath`”（文件路径）似乎是一个更好的名称。本节将向大家完整地例示如何使用这个类，其中包括相关的 `FilenameFilter`（文件名过滤器）接口。

1.1: 创建一个新的 `File` 对象

`File` 类提供了若干处理文件和获取它们基本信息的方法。

```
File myFile;
myFile = new File("mymotd");
myFile = new File("/", "mymotd");
// more useful if the directory or filename is
// a variable
File myDir = new File("/");
myFile = new File(myDir, "mymotd");
```

你所使用的构造函数经常取决于你所使用的其他文件对象。例如，如果你在你的应用程序中只使用一个文件，那么就会使用第一个构造函数。如果你使用一个公共目录中的若干文件，那么使用第二个或者第三个构造函数可能更容易。

`File` 类提供了独立于平台的方法来操作由本地文件系统维护的文件。然而它不允许你存取文件的内容。

注 — 你可以使用一个 `File` 对象来代替一个 `String` 作为 `FileInputStream` 和 `FileOutputStream` 对象的构造函数参数。这是一种推荐方法，因为它独立于本地文件系统的约定。

1.2: 目录列表器

`File` 类并不仅仅是对现有目录路径、文件或者文件组的一个表示。亦可用一个 `File` 对象新建一个目录，甚至创建一个完整的目录路径——假如它尚不存在的话。亦可用它了解文件的属性（长度、上一次修改日期、读 / 写属性等），检查一个 `File` 对象到底代表一个文件还是一个目录，以及删除一个文件等等。下列程序完整展示了如何运用 `File` 类剩下的这些方法：

```
// MakeDirectories.java
import java.io.*;

public class MakeDirectories {
    private final static String usage =
        "Usage:MakeDirectories path1 ...\n" +
        "Creates each path\n" +
        "Usage:MakeDirectories -d path1 ...\n" +
        "Deletes each path\n" +
        "Usage:MakeDirectories -r path1 path2\n" +
```

```
"Renames from path1 to path2\n";
private static void usage() {
    System.err.println(usage);
    System.exit(1);
}
private static void fileData(File f) {
    System.out.println(
        "Absolute path: " + f.getAbsolutePath() +
        "\n Can read: " + f.canRead() +
        "\n Can write: " + f.canWrite() +
        "\n getName: " + f.getName() +
        "\n getParent: " + f.getParent() +
        "\n getPath: " + f.getPath() +
        "\n length: " + f.length() +
        "\n lastModified: " + f.lastModified());
    if(f.isFile())
        System.out.println("it's a file");
    else if(f.isDirectory())
        System.out.println("it's a directory");
}
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
    for( ; count < args.length; count++) {
        File f = new File(args[count]);
        if(f.exists()) {
            System.out.println(f + " exists");
            if(del) {
                System.out.println("deleting..." + f);
            }
        }
    }
}
```

```
        f.delete();
    }
}
else { // Doesn't exist
    if(!del) {
        f.mkdirs();
        System.out.println("created " + f);
    }
}
fileData(f);
}
}
```

在 `fileData()` 中, 可看到应用了各种文件调查方法来显示与文件或目录路径有关的信息。`main()` 应用的第一个方法是 `renameTo()`, 利用它可以重命名 (或移动) 一个文件至一个全新的路径 (该路径由参数决定), 它属于另一个 `File` 对象。这也适用于任何长度的目录。

若试验上述程序, 就可发现自己能制作任意复杂程度的一个目录路径, 因为 `mkdirs()` 会帮我们完成所有工作。

1.3: 文件测试和工具

当你创建一个 `File` 对象时, 你可以使用下面任何一种方法来获取有关文件的信息:

文件名

- | `String getName()`
- | `String getPath()`
- | `String getAbsolutePath()`
- | `String getParent()`
- | `boolean renameTo(File newName)`

文件测试

- | `boolean exists()`
- | `boolean canWrite()`
- | `boolean canRead()`
- | `boolean isFile()`
- | `boolean isDirectory()`
- | `boolean isAbsolute()`

通用文件信息和工具

- | `long lastModified()`
- | `long length()`
- | `boolean delete()`

目录工具

- | boolean mkdir()
- | String[] list()

1.4: 随机存取文件

你经常会发现你只想读取文件的一部分数据,而不需要从头至尾读取整个文件。你可能想访问一个作为数据库的文本文件,此时你会移动到某一条记录并读取它的数据,接着移动到另一个记录,然后再到其他记录——每一条记录都位于文件的不同部分。Java 编程语言提供了一个 `RandomAccessFile` 类来处理这种类型的输入输出。

你可以用如下两种方法来打开一个随机存取文件:

- | 用文件名
`myRAFile = new RandomAccessFile(String name, String mode);`
- | 用文件对象
`myRAFile = new RandomAccessFile(File file, String mode);`

`mode` 参数决定了你对这个文件的存取是只读(r)还是读/写(rw)。

例如,你可以打开一个数据库文件并准备更新:

```
RandomAccessFile myRAFile;  
myRAFile = new RandomAccessFile("db/stock.dbf","rw");
```

存取信息

`RandomAccessFile` 对象按照与数据输入输出对象相同的方式来读写信息。你可以访问在 `DataInputStream` 和 `DataOutputStream` 中所有的 `read()`和 `write()`操作。

Java 编程语言提供了若干种方法,用来帮助你在文件中移动。

- | `long getFilePointer();`
返回文件指针的当前位置。
- | `void seek(long pos);`
设置文件指针到给定的绝对位置。这个位置是按照从文件开始的字节偏移量给出的。位置 0 标志文件的开始。
- | `long length()`
返回文件的长度。位置 `length()`标志文件的结束。

添加信息

你可以使用随机存取文件来得到文件输出的添加模式。

```
myRAFile = new RandomAccessFile("java.log","rw");  
myRAFile.seek(myRAFile.length());  
// Any subsequent write(s) will be appended to the file
```

2: 过滤器

`FilterInputStream` 和 `FilterOutputStream` (这两个名字不十分直观) 提供了相应的装饰器接口,用于控制一个特定的输入流 (`InputStream`) 或者输出流 (`OutputStream`)。它们分别是 `InputStream` 和 `OutputStream` 衍生出来的。此外,它们都属于抽象类,在理论上为我们与一个流的不同通信手段都提供了一个通用的接口。事实上, `FilterInputStream` 和

FilterOutputStream 只是简单地模仿了自己的基础类，它们是一个装饰器的基本要求。

2.1 通过 FilterInputStream 从 InputStream 里读入数据

FilterInputStream 类要完成两件全然不同的事情。其中，DataInputStream 允许我们读取不同的基本类型数据以及 String 对象（所有方法都以“read”开头，比如 readByte(), readFloat() 等等）。伴随对应的 DataOutputStream，我们可通过数据“流”将基本类型的数据从一个地方搬到另一个地方。若读取块内的数据，并自己进行解析，就不需要用到 DataInputStream。但在其他许多情况下，我们一般都想用它对自己读入的数据进行自动格式化。

剩下的类用于修改 InputStream 的内部行为方式：是否进行缓冲，是否跟踪自己读入的数据行，以及是否能够推回一个字符等等。后两种类看起来特别象提供对构建一个编译器的支持（换言之，添加它们为了支持 Java 编译器的构建），所以在常规编程中一般都用不着它们。

也许几乎每次都要缓冲自己的输入，无论连接的是哪个 IO 设备。所以 IO 库最明智的做法就是将未缓冲输入作为一种特殊情况处理，同时将缓冲输入接纳为标准做法。

2.2 通过 FilterOutputStream 向 OutputStream 里写入数据

与 DataInputStream 对应的是 DataOutputStream，后者对各个基本数据类型以及 String 对象进行格式化，并将其置入一个数据“流”中，以便任何机器上的 DataInputStream 都能正常地读取它们。所有方法都以“write”开头，例如 writeByte(), writeFloat() 等等。

若想进行一些真正的格式化输出，比如输出到控制台，请使用 PrintStream。利用它可以打印出所有基本数据类型以及 String 对象，并可采用一种易于查看的格式。这与 DataOutputStream 正好相反，后者的目标是将那些数据置入一个数据流中，以便 DataInputStream 能够方便地重新构造它们。System.out 静态对象是一个 PrintStream。

PrintStream 内两个重要的方法是 print() 和 println()。它们已进行了覆盖处理，可打印出所有数据类型。print() 和 println() 之间的差异是后者在操作完毕后会自动添加一个新行。

BufferedOutputStream 属于一种“修改器”，用于指示数据流使用缓冲技术，使自己不必每次都向流内物理性地写入数据。通常都应将它应用于文件处理和控制器 IO。

三：I/O 流的基本应用

1: 输入流

1.1: 缓冲的输入文件

程序示例


```
import java.io.*;
public class Test
{
    public static void main(String args[])
    {
        String currentLine;
        try
        {
            DataInputStream in =
                new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream("Test.java")
                    )
                );
            while ((currentLine = in.readLine()) != null)
                System.out.println(currentLine);
        }
        catch (IOException e)
        {
            System.err.println("Error: " + e);
        } // End of try/catch structure.
    } // End of method: main
} // End of class
```

为打开一个文件以便输入,需要使用一个 `FileInputStream`,同时将一个 `String` 或 `File` 对象作为文件名使用。为提高速度,最好先对文件进行缓冲处理,从而获得用于一个 `BufferedInputStream` 的构建器的结果句柄。为了以格式化的形式读取输入数据,我们将那个结果句柄赋给用于一个 `DataInputStream` 的构建器。`DataInputStream` 是我们的最终 (`final`) 对象,并是我们进行读取操作的接口。

在这个例子中,只用到了 `readLine()` 方法,但理所当然任何 `DataInputStream` 方法都可以采用。一旦抵达文件末尾, `readLine()` 就会返回一个 `null` (空),以便中止并退出 `while` 循环。

1.2: 格式化内存输入

`StringBufferInputStream` 的接口是有限的,所以通常需要将其封装到一个 `DataInputStream` 内,从而增强它的能力。然而,若选择用 `readByte()` 每次读出一个字符,那么所有值都是有效的,所以不可再用返回值来侦测何时结束输入。相反,可用 `available()` 方法判断有多少字符可用。下面这个例子展示了如何从文件中一次读出一个字符:

```
// TestEOF.java
import java.io.*;
```

```
public class TestEOF {
    public static void main(String[] args) {
        try {
            DataInputStream in =
                new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream("TestEof.java")));
            while(in.available() != 0)
                System.out.print((char)in.readByte());
        } catch (IOException e) {
            System.err.println("IOException");
        }
    }
}
```

注意取决于当前从什么媒体读入, `available()` 的工作方式也是有所区别的。它在字面上意味着“可以不受阻塞读取的字节数量”。对一个文件来说, 它意味着整个文件。但对一个不同种类的数据流来说, 它却可能有不同的含义。因此在使用时应考虑周全。

为了在这样的情况下侦测输入的结束, 也可以通过捕获一个违例来实现。然而, 若真的用违例来控制数据流, 却显得有些大材小用。

2 输出流

两类主要的输出流是按它们写入数据的方式划分的: 一种按人的习惯写入, 另一种为了以后由一个 `DataInputStream` 而写入。 `RandomAccessFile` 是独立的, 尽管它的数据格式兼容于 `DataInputStream` 和 `DataOutputStream`。

2.1: 保存与恢复数据

`PrintStream` 能格式化数据, 使其能按我们的习惯阅读。但为了输出数据, 以便由另一个数据流恢复, 则需用一个 `DataOutputStream` 写入数据, 并用一个 `DataInputStream` 恢复(获取)数据。当然, 这些数据流可以是任何东西, 但这里采用的是一个文件, 并进行了缓冲处理, 以加快读写速度。

注意字串是用 `writeBytes()` 写入的, 而非 `writeChars()`。若使用后者, 写入的就是 16 位 Unicode 字符。由于 `DataInputStream` 中没有补充的“`readChars`”方法, 所以不得不用 `readChar()` 每次取出一个字符。所以对 ASCII 来说, 更方便的做法是将字符作为字节写入, 在后面跟随一个新行; 然后再用 `readLine()` 将字符当作普通的 ASCII 行读回。

`writeDouble()` 将 `double` 数字保存到数据流中, 并用补充的 `readDouble()` 恢复它。但为了保证任何读方法能够正常工作, 必须知道数据项在流中的准确位置, 因为既有可能将保存的 `double` 数据作为一个简单的字节序列读入, 也有可能作为 `char` 或其他格式读入。所以必须要么为文件中的数据采用固定的格式, 要么将额外的信息保存到文件中, 以便正确判断数据的存放位置。

2.2: 读写随机访问文件

示例代码

```
import java.io.*;
public class Test
{
    public static void main(String args[])
    {
        try
        {
            RandomAccessFile f =new RandomAccessFile( "test.txt","rw");
            int i;
            double d;
            for (i= 0; i< 10; i++){
                f.writeDouble(3.14f*i);
            }
            f.seek(16);
            f.writeDouble(0);
            f.seek(0);
            for (i= 0; i< 10; i++)
            {
                d=f.readDouble();
                System.out.println("[ "+i+"]: "+d);
            }
            f.close();
        }
        catch (IOException io)
        {
            System.out.println(io);
            System.exit(-1);
        }
    } // End of try/catch structure.
} // End of method: main
} // End of class:
```

正如早先指出的那样,RandomAccessFile 与 IO 层次结构的剩余部分几乎是完全隔离的,尽管它也实现了 DataInput 和 DataOutput 接口。所以不可将其与 InputStream 及 OutputStream 子类的任何部分关联起来。尽管也许能将一个 ByteArrayInputStream 当作一个随机访问元素对待,但只能用 RandomAccessFile 打开一个文件。必须假定 RandomAccessFile 已得到了正确的缓冲,因为我们不能自行选择。

可以自行选择的是第二个构建器参数:可决定以“只读”(r)方式或“读写”(rw)方式打开一个 RandomAccessFile 文件。

使用 RandomAccessFile 的时候,类似于组合使用 DataInputStream 和 DataOutputStream (因为它实现了等同的接口)。除此以外,还可看到程序中使用了 seek(),以便在文件中到处移动,对某个值作出修改。

3: 快捷文件处理

由于以前采用的一些典型形式都涉及到文件处理,所以大家也许会怀疑为什么要进行那么多的代码输入——这正是装饰器方案一个缺点。本部分将向大家展示如何创建和使用典型文件读取和写入配置的快捷版本。为了将每个类都添加到库内,只需将其置入适当的目录,并添加对应的 package 语句即可。

3.1: 快速文件输入

若想创建一个对象,用它从一个缓冲的 `DataInputStream` 中读取一个文件,可将这个过程封装到一个名为 `InFile` 的类内。如下所示:

```
// InFile.java
import java.io.*;

public class InFile extends DataInputStream {
    public InFile(String filename)
        throws FileNotFoundException {
        super(
            new BufferedInputStream(
                new FileInputStream(filename)));
    }
    public InFile(File file)
        throws FileNotFoundException {
        this(file.getPath());
    }
}
```

无论构建器的 `String` 版本还是 `File` 版本都包括在内,用于共同创建一个 `FileInputStream`。

就象这个例子展示的那样,现在可以有效减少创建文件时由于重复强调造成的问题。

3.2: 快速输出格式化文件

亦可用同类型的方法创建一个 `PrintStream`, 令其写入一个缓冲文件。

```
// PrintFile.java
import java.io.*;

public class PrintFile extends PrintStream {
    public PrintFile(String filename)
        throws IOException {
        super(
            new BufferedOutputStream(
                new FileOutputStream(filename)));
    }
}
```

```
}  
public PrintFile(File file)  
    throws IOException {  
    this(file.getPath());  
}  
}
```

注意构建器不可能捕获一个由基础类构建器“掷”出的违例。

3.3: 快速输出数据文件

最后, 利用类似的快捷方式可创建一个缓冲输出文件, 用它保存数据 (与由人观看的数据格式相反):

```
// OutFile.java  
package com.bruceeckel.tools;  
import java.io.*;  
  
public class OutFile extends DataOutputStream {  
    public OutFile(String filename)  
        throws IOException {  
        super(  
            new BufferedOutputStream(  
                new FileOutputStream(filename)));  
    }  
    public OutFile(File file)  
        throws IOException {  
        this(file.getPath());  
    }  
}
```

4: 从标准输入中读取数据

以 Unix 首先倡导的“标准输入”、“标准输出”以及“标准错误输出”概念为基础, Java 提供了相应的 `System.in`, `System.out` 以及 `System.err`。贯这一整本书, 大家都会接触到如何用 `System.out` 进行标准输出, 它已预封装成一个 `PrintStream` 对象。`System.err` 同样是一个 `PrintStream`, 但 `System.in` 是一个原始的 `InputStream`, 未进行任何封装处理。这意味着尽管能直接使用 `System.out` 和 `System.err`, 但必须事先封装 `System.in`, 否则不能从中读取数据。

典型情况下, 我们希望用 `readLine()` 每次读取一行输入信息, 所以需要将 `System.in` 封装到一个 `DataInputStream` 中。这是 Java 1.0 进行行输入时采取的“老”办法。在本章稍后, 大家还会看到 Java 1.1 的解决方案。下面是个简单的例子, 作用是回应我们键入的每一行内容:

```
// Echo.java
import java.io.*;

public class Echo {
    public static void main(String[] args) {
        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(System.in));
        String s;
        try {
            while((s = in.readLine()).length() != 0)
                System.out.println(s);
            // An empty line terminates the program
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

之所以要使用 try 块,是由于 readLine() 可能“掷”出一个 IOException。注意同其他大多数流一样,也应对 System.in 进行缓冲。

由于在每个程序中都要将 System.in 封装到一个 DataInputStream 内,所以显得有点不方便。但采用这种设计方案,可以获得最大的灵活性。

练习实践课:

本章内容为 IO 操作, 实践重点:

- I IO 基础类
- I 文件操作, 包括文件的建立、写入、读出、删除等

程序 1:

文件操作

需求: 文件的建立、写入、读出、删除。

目标:

- 1、目录的建立;
- 2、文件的建立;
- 3、文件内容写入;
- 4、文件内容读取。

程序:

```
//: FileProcess.java
package com.useful.java.part6;

import java.io.*;

public class FileProcess{

    public FileProcess(){

    }

    public static void main(String[] args){
        FileProcess process=new FileProcess();
        String dirname="c:/testdir";
        String filename="a.txt";
        process.createDir(dirname);
        process.createFile(dirname+"/"+filename);
        process.writeFile(dirname+"/"+filename);
        process.readFile(dirname+"/"+filename);
        process.writeTxt(dirname+"/testtext.txt");
    }

    public void createDir(String dirname){
        File file=new File(dirname);
        if(file.exists()==false){
            file.mkdirs();
        }
    }
}
```

```
public void createFile(String filename){
    try{
        File file=new File(filename);
        file.createNewFile();
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void writeFile(String filename){
    try{
        File file=new File(filename);
        FileOutputStream output=new FileOutputStream(file);
        output.write("i am test.".getBytes());
        output.close();
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void readFile(String filename){
    try{
        File file=new File(filename);
        FileInputStream input=new FileInputStream(file);
        byte byteValues[]=new byte[(int)file.length()];
        input.read(byteValues);
        input.close();
        System.out.println("file content is "+(new String(byteValues)));
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void writeTxt(String filename){
    try{
        File file=new File(filename);
        Writer writer=new FileWriter(file);
        writer.write("Holen ,Holen");
        writer.close();
    }
}
```



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

说明:

1、本程序将在 C 盘根目录下, 建立 testdir 目录, 并在此目录下建立两个文本文件; 注意上例中, 往文件中写入内容的方式, 一种是用流的方式, 一种是文本方式。

作业

1: 利用输入输出流编写一个程序, 可以实现文件复制的功能, 程序的命令行参数的形式及操作功能均类似于 DOS 中的 copy 命令

2: 为前面完成的地址本程序添加文件的操作, 把存放在内存集合中的数据存放到文件中, 形成用 Swing 完成表现层, 集合完成逻辑层, I/O 完成数据存储的结构。

第十三章 多线程

教学目标:

- i 掌握线程的基本概念
- i 掌握多线程的编写□
- i 掌握多线程的控制
- i 掌握多线程的同步

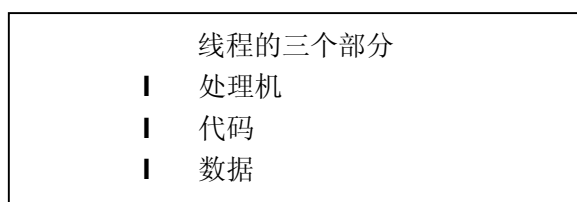
一：线程的基本概念

1: 什么是线程

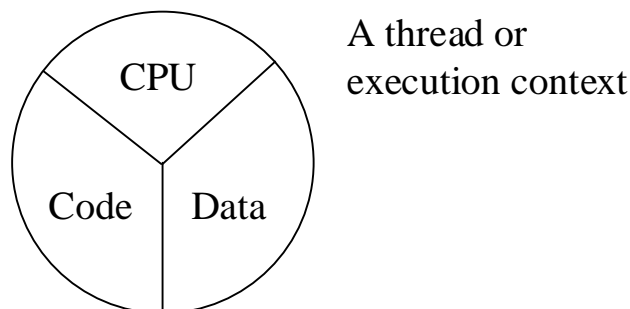
一个关于计算机的简化的视图是：它有一个执行计算的处理器、包含处理器所执行的程序的 ROM(只读存储器)、包含程序所要操作的数据的 RAM(只读存储器)。在这个简化视图中，只能执行一个作业。一个关于现代计算机比较完整的视图是允许计算机在同一个时间执行一个以上的作业。

你不需关心这一点是如何实现的，只需从编程的角度考虑就可以了。如果你要执行一个以上的作业，这类似有一台以上的计算机。在这个模型中，**线程，被认为是带有自己的程序代码和数据的虚拟处理机的封装。** `java.lang.Thread` 类允许用户创建并控制他们的线程。

2: 线程的三个部分



进程是正在执行的程序。一个或更多的线程构成了一个进程。一个线程或执行上下文由三个主要部分组成



- | 一个虚拟处理器
- | CPU 执行的代码
- | 代码操作的数据

代码可以或不可以由多个线程共享，这和数据是独立的。两个线程如果执行同一个类的实例代码，则它们可以共享相同的代码。

类似地，数据可以或不可以由多个线程共享，这和代码是独立的。两个线程如果共享对一个公共对象的存取，则它们可以共享相同的数据。

在 Java 编程中，虚拟处理器封装在 `Thread` 类的一个实例里。构造线程时，定义其上下文的代码和数据是由传递给它的构造函数的对象指定的。

3: Java 中的线程的概念

几乎每种操作系统都支持进程的概念——进程就是在某种程度上相互隔离的、独立运行的程序。

线程化是允许多个活动共存于一个进程中的工具。大多数现代的操作系统都支持线程,而且线程的概念以各种形式已存在了好多年。Java 是第一个在语言本身中显式地包含线程的主流编程语言,它没有把线程化看作是底层操作系统的工具。

有时候,线程也称作轻量级进程。就象进程一样,线程在程序中是独立的、并发的执行路径,每个线程有它自己的堆栈、自己的程序计数器和自己的局部变量。但是,与分隔的进程相比,进程中的线程之间的隔离程度要小。它们共享内存、文件句柄和其它每个进程应有的状态。进程可以支持多个线程,它们看似同时执行,但互相之间并不同步。一个进程中的多个线程共享相同的内存地址空间,这就意味着它们可以访问相同的变量和对象,而且它们从同一堆中分配对象。尽管这让线程之间共享信息变得更容易,但您必须小心,确保它们不会妨碍同一进程里的其它线程。

Java 线程工具和 API 看似简单。但是,编写有效使用线程的复杂程序并不十分容易。因为有多个线程共存在相同的内存空间中并共享相同的变量,所以您必须小心,确保您的线程不会互相干扰。

每个 Java 程序都至少有一个线程 — 主线程。当一个 Java 程序启动时,JVM 会创建主线程,并在该线程中调用程序的 `main()` 方法。

JVM 还创建了其它线程,您通常都看不到它们 — 例如,与垃圾收集、对象终止和其它 JVM 内务处理任务相关的线程。其它工具也创建线程,如 AWT (抽象窗口工具箱 (Abstract Windowing Toolkit)) 或 Swing UI 工具箱、servlet 容器、应用程序服务器和 RMI (远程方法调用 (Remote Method Invocation))。

在 Java 程序中使用线程有许多原因。如果您使用 Swing、servlet、RMI 或 Enterprise JavaBeans (EJB) 技术,您也许没有意识到您已经在使用线程了。

使用线程的一些原因是它们可以帮助:

使 UI 响应更快

事件驱动的 UI 工具箱 (如 AWT 和 Swing) 有一个事件线程,它处理 UI 事件,如击键或鼠标点击。

利用多处理器系统

多处理器 (MP) 系统比过去更普及了。以前只能在大型数据中心和科学计算设施中才能找到它们。现在许多低端服务器系统 — 甚至是一些台式机系统 — 都有多个处理器。

调度的基本单位通常是线程;如果某个程序只有一个活动的线程,它一次只能在一个处理器上运行。如果某个程序有多个活动线程,那么可以同时调度多个线程。在精心设计的程序中,使用多个线程可以提高程序吞吐量和性能。

简化建模

在某些情况下,使用线程可以使程序编写和维护起来更简单。考虑一个仿真应用程序,您要在其中模拟多个实体之间的交互作用。给每个实体一个自己的线程可以使许多仿真和对应用程序的建模大大简化。

服务器应用程序从远程来源 (如套接字) 获取输入。当读取套接字时,如果当前没有可用数据,那么对 `SocketInputStream.read()` 的调用将会阻塞,直到有可用数据为止。

如果单线程程序要读取套接字,而套接字另一端的实体并未发送任何数据,那么该程序只会永远等待,而不执行其它处理。相反,程序可以轮询套接字,查看是否有可用数据,但通常不会使用这种做法,因为会影响性能。

线程和进程的区别是:

每个进程都有独立的代码和数据空间(进程上下文), 进程切换的开销大。

线程作为轻量的进程, 同一类线程可以共享代码和数据空间, 但每个线程有独立的运行栈和程序计数器, 因此线程切换的开销较小。

多进程——在操作系统中能同时运行多个任务(程序), 也称多任务。

多线程——在同一应用程序中有多个顺序流同时执行。

4: Java 编程中的线程

4.1 第一个线程

下面来学习如何创建第一个线程, 以及如何使用构造函数参数来为一个线程提供运行时的数据和代码。

一个 Thread 类构造函数带有一个参数, 它是 Runnable 的一个实例。一个 Runnable 是由一个实现了 Runnable 接口(即, 提供了一个 public void run()方法)的类产生的。

例如:

```
1. public class ThreadTest {
2. public static void main(String args[]) {
3.     XYZ r = new XYZ();
4.     Thread t = new Thread(r);
5. }
6. }
7.
8. class XYZ implements Runnable {
9.     int i;
10.
11.     public void run() {
12.         while (true) {
13.             System.out.println("Hello " + i++);
14.             if (i == 50) break;
15.         }
16.     }
17. }
```

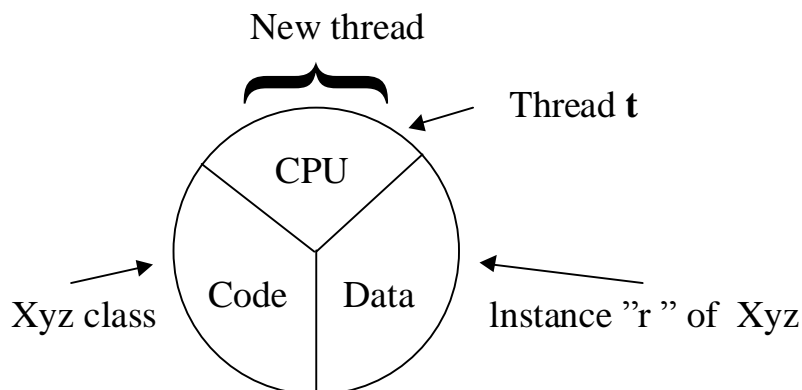
首先, main()方法构造了 XYZ 类的一个实例 r。实例 r 有它自己的数据, 在这里就是整数 i。因为实例 r 是传给 Thread 的类构造函数的, 所以 r 的整数 i 就是线程运行时刻所操作的数据。线程总是从它所装载的 Runnable 实例(在本例中, 这个实例就是 r。)的 run()方法开始运行。

一个多线程编程环境允许创建基于同一个 Runnable 实例的多个线程。这可以通过以下方法来做到:

```
Thread t1= new Thread(r);
```

```
Thread t2= new Thread(r);
```

此时, 这两个线程共享数据和代码。



总之, 线程通过 Thread 对象的一个实例引用。线程从装入的 Runnable 实例的 run() 方法开始执行。线程操作的数据从传递给 Thread 构造函数的 Runnable 的特定实例处获得。

4.2 启动线程

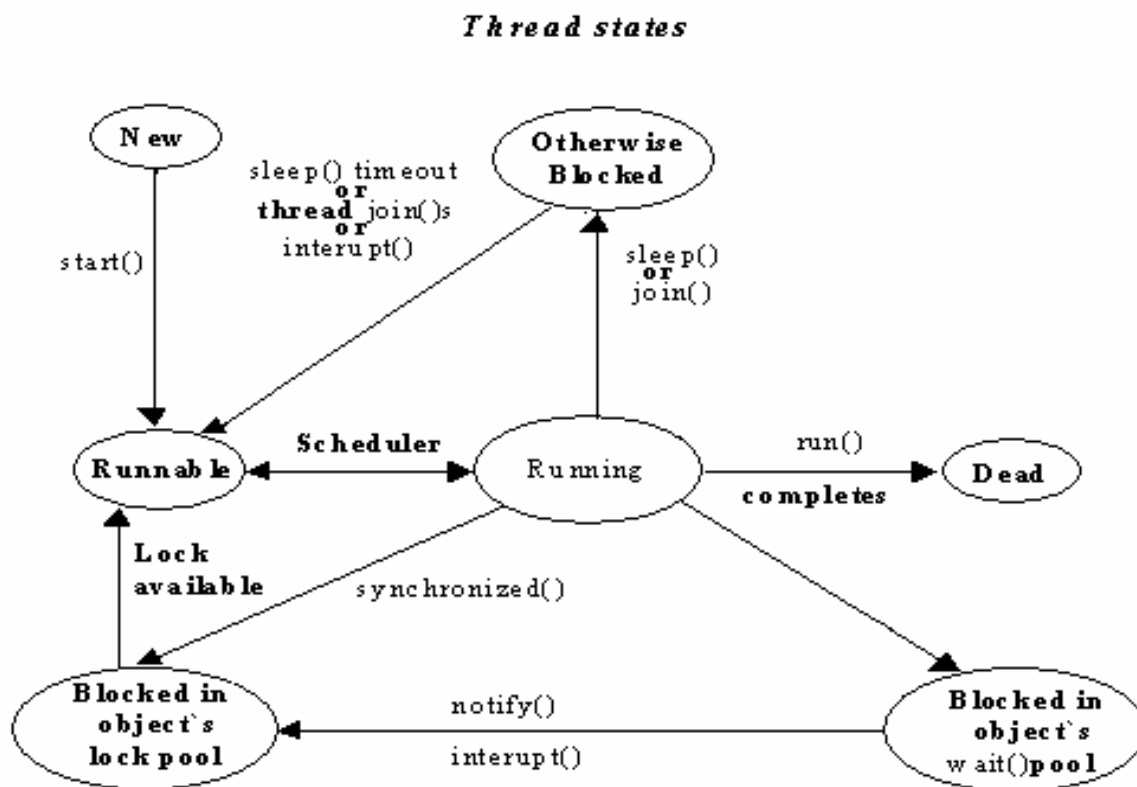
一个新创建的线程并不自动开始运行。你必须调用它的 start() 方法。例如, 你可以发现上例中第 4 行代码中的命令:

```
t.start();
```

调用 start() 方法使线程所代表的虚拟处理机处于可运行状态, 这意味着它可以由 JVM 调度并执行。这并不意味着线程就会立即运行。

4.3 线程状态

一个 Thread 对象在它的生命周期中会处于各种不同的状态。下图形象地说明了这点:



线程状态及状态转换规则具体说明如下:

(1) 新建状态——就绪状态:

一个新创建的线程(使用 new+Thread 构造方法创建的对象)不会自动运行,此时处于新建(New)状态。当程序员显式调用线程的 start() 方法时,该线程进入就绪(Runnable)状态,也称可运行状态。进入就绪状态的线程不一定立即开始运行,因为此时计算机 CPU 可能正在运行其它的线程。可能有多个线程同时进入就绪状态,在就绪队列中排队等候。

(2) 就绪状态——运行状态:

Java 运行时系统提供的线程调度器按照一定的规则进行调度,一旦某个线程获得执行机会,则立即进入运行(Running)状态、开始执行线程体代码。

(3) 运行状态——阻塞状态:

处于运行状态的线程可能因某种事件的发生而进入阻塞(Blocked)状态、暂时停止执行。例如,线程进行 I/O 操作,等待用户输入数据。当一个运行状态的线程发生阻塞时,调度器立即调度就绪队列中的另一个线程开始运行。

(4) 阻塞状态——就绪状态:

当处于阻塞状态的线程所等待的条件已经具备,例如用户输入操作已经完成时,该线程将解除阻塞,进入就绪状态。注意,不是恢复执行,而是重新到就绪队列中去排队。

(5) 运行状态——终止状态:

线程的 run() 方法正常执行完毕后,其运行也就自然结束,线程进入终止(Dead)状态。也可以在运行过程中,非正常地终止一个线程的执行,例如调用其 stop() 方法。处于终止状态的线程不能在重新运行,因此不允许在一个 Thread 对象上两次调用 start() 方法。

二: 多线程的编写和控制

1: 多线程的创建

1.1: 实现 Runnable 接口

Java 中引入线程机制的目的在于实现多线程,以提高程序的效能。这主要是通过多线程之间共享代码和数据来实现的,例如可以使用同一个 Runnable 接口(的实现类)类型的实例构造多个线程。

示例 13-1 使用多线程

程序: TestThread2.java

```
public class TestThread2 {
    public static void main(String args[]) {
        Runner2 r = new Runner2();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
```

```
}

class Runner2 implements Runnable {
    public void run() {
        for(int i=0; i<20; i++) {
            String s = Thread.currentThread().getName();
            System.out.println(s + ": " + i);
        }
    }
}
```

程序运行结果为:

程序 TestThread2.java 中创建了两个新的线程 t1 和 t2, 他们共享代码——Runner2 中的 run() 方法, 同时也共享数据——Runnable 类型的对象 r, 两个线程在运行过程中分别操纵对象 r 调用其 run() 方法。从输出结果可以看出, 线程 t1 和 t2 作为独立的顺序控制流, 并发地交替执行。可以想象, 如果先启动的线程因某种原因处于阻塞状态, 例如等待用户键盘输入数据, CPU 会立即转而执行其他的线程, 而不必空置。

需要注意的是, 这和在 main() 方法中直接调用两个方法有本质的不同, 那样不会出现交替的情况, 必须要前面的方法执行完才会执行后面的方法。

1.2: 继承 Thread

在示例 13-1 中, 直接定义了 Runnable 接口的实现类来提供线程体, 这是创建线程的基本方式。除此之外, 还可以采用直接继承 Thread 类、重写其中的 run() 方法并以之作为线程体的方式创建线程, 见示例 13-2:

程序: TestThread3.java

```
public class TestThread3 {
    public static void main(String args[]){
        Thread t = new Runner3();
        t.start();
    }
}

class Runner3 extends Thread {
    public void run() {
        for(int i=0; i<30; i++) {
            System.out.println("No. " + i);
        }
    }
}
```


程序 TestThread3.java 与示例 13-1 中的 TestThread1.java 运行结果相同。从中可以看出, 第二种创建线程方式的一般步骤为:

1. 定义一个类继承 Thread 类, 重写其中的 run() 方法;
2. 创建该 Thread 子类的对象;
3. 调用该对象的 start() 方法, 启动线程。

这种情况下, 线程 t 所执行的线程体就是 t 本身的成员方法 run()。实际上, Thread 类也已经实现了 Runnable 接口。

1.3: 两种方法的比较

给定各种方法的选择, 你如何决定使用哪个? 每种方法都有若干优点。

实现 Runnable 的优点

从面向对象的角度来看, Thread 类是一个虚拟处理机严格的封装, 因此只有当处理机模型修改或扩展时, 才应该继承类。正因为这个原因和区别一个正在运行的线程的处理机、代码和数据部分的意义, 本教程采用了这种方法。

由于 Java 技术只允许单一继承, 所以如果你已经继承了 Thread, 你就不能再继承其它任何类, 例如 Applet。在某些情况下, 这会使你只能采用实现 Runnable 的方法。

因为有时你必须实现 Runnable, 所以你可能喜欢保持一致, 并总是使用这种方法。

继承 Thread 的优点

当一个 run() 方法体现在继承 Thread 类的类中, 用 this 指向实际控制运行的 Thread 实例。因此, 代码不再需要使用如下控制:

```
Thread.currentThread().join();  
而可以简单地用:  
join();
```

因为代码简单了一些, 许多 Java 编程语言的程序员使用扩展 Thread 的机制。注意: 如果你采用这种方法, 在你的代码生命周期的后期, 单继承模型可能会给你带来困难。

2: 线程调度

尽管线程变为可运行的, 但它并不立即开始运行。在一个只带有一个处理机的机器上, 在一个时刻只能进行一个动作。下节描述了如果有一个以上可运行线程时, 如何分配处理机。

在 Java 中, 线程是**抢占式**的, 但并不一定是分时的 (一个常见的错误是认为“抢占式”只不过是“分时”的一种新奇的称呼而已)。

抢占式调度模型是指可能有多个线程是可运行的, 但只有一个线程在实际运行。这个线程会一直运行, 直至它不再是可运行的, 或者另一个具有更高优先级的线程成为可运行的。对于后面一种情形, 低优先级线程被高优先级线程抢占了运行的机会。

下面几种情况下, 当前线程会放弃 CPU, 进入阻塞状态:

- I 线程调用了 `yield()`, `suspend()` 或 `sleep()` 方法主动放弃;
- I 由于当前线程进行 I/O 访问, 外存读写, 等待用户输入等操作, 导致线程阻塞;
- I 为等候一个条件变量, 线程调用 `wait()` 方法;
- I 抢先式系统下, 有高优先级的线程参与调度; 时间片方式下, 当前时间片用完, 有同优先级的线程参与调度。

2.1: 线程优先级

每个线程都有自己的优先级, 通常优先级高的线程将先于优先级低的线程执行。线程的优先级用数字来表示, 范围从 1 到 10, 主线程的缺省优先级是 5。其他线程的优先级默认与父线程相同, 也可以使用 `Thread` 类的下述方法获得或设置线程对象的优先级:

```
public final int getPriority();
```

```
public final void setPriority(int newPriority)
```

为使用方便, `Thread` 类还提供了几个 `public static int` 常量:

```
Thread.MIN_PRIORITY = 1
```

```
Thread.MAX_PRIORITY = 10
```

```
Thread.NORM_PRIORITY = 5
```

示例 13-3 使用线程优先级程序: `TestPriority.java`

```
public class TestPriority {
    public static void main(String args[]){
        System.out.println("线程名\t优先级");
        Thread current = Thread.currentThread();
        System.out.print(current.getName() + "\t");
        System.out.println(current.getPriority());
        Thread t1 = new Runner();
        Thread t2 = new Runner();
        Thread t3 = new Runner();
        t1.setName("First");
        t2.setName("Second");
        t3.setName("Third");
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
    }  
}  
  
class Runner extends Thread {  
    public void run() {  
        System.out.print(this.getName() + "\t");  
        System.out.println(this.getPriority());  
    }  
}
```

程序运行结果为:

线程名	优先级
main	5
Second	10
First	5
Third	5

一个线程可能因为各种原因而不再是可运行的。线程的代码可能执行了一个 `Thread.sleep()` 调用, 要求这个线程暂停一段固定的时间。这个线程可能在等待访问某个资源, 而且在这个资源可访问之前, 这个线程无法继续运行。

所有可运行线程根据优先级保存在池中。当一个被阻塞的线程变成可运行时, 它会被放回相应的可运行池。优先级最高的非空池中的线程会得到处理机时间(被运行)。

因为 Java 线程不一定是分时的, 所以你必须确保你的代码中的线程会不时地给另外一个线程运行的机会。这可以通过在各种时间间隔中发出 `sleep()` 调用来做到。

```
1. public class Xyz implements Runnable {  
2. public void run() {  
3. while (true) {  
4. // do lots of interesting stuff  
5. :  
6. // Give other threads a chance  
7. try {  
8. Thread.sleep(10);  
9. } catch (InterruptedException e) {  
10. // This thread's sleep was interrupted  
11. // by another thread  
12. }  
13. }  
14. }  
15. }
```

注意 try 和 catch 块的使用。Thread.sleep() 和其它使线程暂停一段时间的方法是可中断的。线程可以调用另外一个线程的 interrupt() 方法, 这将向暂停的线程发出一个 InterruptedException。

注意 Thread 类的 sleep() 方法对当前线程操作, 因此被称作 Thread.sleep(x), 它是一个静态方法。sleep() 的参数指定以毫秒为单位的线程最小休眠时间。除非线程因为中断而提早恢复执行, 否则它不会在这段时间之前恢复执行。

Thread 类的另一个方法 yield(), 可以用来使具有相同优先级的线程获得执行的机会。如果具有相同优先级的其它线程是可运行的, yield() 将把调用线程放到可运行池中并使另一个线程运行。如果没有相同优先级的可运行进程, yield() 什么都不做。

注意 sleep() 调用会给较低优先级线程一个运行的机会。yield() 方法只会给相同优先级线程一个执行的机会。

3: 线程控制

为有效地进行线程管理和状态控制, Object 类和 Thread 类中提供多个有用的方法, 如表 13-1 所示:

表 13-1 线程控制基本方法。

方 法	功 能
isAlive()	判断线程是否还“活”着, 即线程是否还未终止。
getPriority()	获得线程的优先级数值
setPriority()	设置线程的优先级数值
Thread.sleep()	将当前线程睡眠指定毫秒数
join()	调用某线程的该方法, 将当前线程与该线程“合并”, 即等待该线程结束, 再恢复当前线程的运行。
yield()	让出 CPU, 当前线程进入就绪队列等待调度。
wait()	当前线程进入对象的 wait pool。
notify()/notifyAll()	唤醒对象的 wait pool 中的一个/所有等待线程。
suspend()	挂起当前线程, 不提倡使用。
resume()	解除当前线程的挂起状态, 不提倡使用。
stop()	终止当前线程的执行, 不提倡使用。

示例 13-4 使用 sleep() 方法

程序: TestSleep.java

```
import java.awt.*;
import java.util.Calendar;
public class TestSleep{
    public static void main(String args[]) {
        Frame f = new Frame("My Watch");
```

```
Label l = new Label();
f.add(l);
f.setSize(100,50);
f.setVisible(true);
while(true){
    Calendar c = Calendar.getInstance();
    l.setText(c.get(Calendar.HOUR_OF_DAY) + ":"
            + c.get(Calendar.MINUTE) + ":"
            + c.get(Calendar.SECOND));
    try{
        Thread.sleep(1000);
    }catch(InterruptedException e){}
}
}
```

本程序实现了电子时钟的功能，其图形界面如图 13-4 所示。

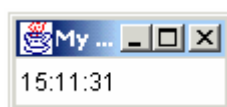


图 13-4 电子时钟

线程阻塞，休眠 1000 毫秒后再恢复运行时刷新显示时间。在其休眠期间，系统可以执行其他程序或线程，以提高运行效率。

示例 使用 join() 方法程序: TestJoin.java

```
public class TestJoin {
    public static void main(String args[]){
        MyRunner r = new MyRunner();
        Thread t = new Thread(r);
        t.start();
        try{
            t.join();
        }catch(InterruptedException e){
        }
        for(int i=0;i<50;i++){
            System.out.println("主线程:" + i);
        }
    }
}
```

```
class MyRunner implements Runnable {  
    public void run() {  
        for(int i=0;i<50;i++) {  
            System.out.println("SubThread: " + i);  
        }  
    }  
}
```

程序运行输出结果为:

SubThread: 0

SubThread: 1

SubThread: 2

.....

SubThread: 49

主线程: 0

主线程: 1

主线程: 2

.....

主线程: 49

从中可以看出, 在主线程执行过程中, 调用 `t.join()` 方法导致当前线程 (主线程) 阻塞, 直到线程 `t` 运行终止后, 主线程才获得执行的机会。如果多线程程序中, 一个线程要用道另一个线程执行后提供的条件, 则可考虑使用 `join()` 方法。

3.1: 创建线程和启动线程并不相同

在一个线程对新线程的 `Thread` 对象调用 `start()` 方法之前, 这个新线程并没有真正开始执行。`Thread` 对象在其线程真正启动之前就已经存在了, 而且其线程退出之后仍然存在。这可以让您控制或获取关于已创建的线程的信息, 即使线程还没有启动或已经完成了。

通常在构造器中通过 `start()` 启动线程并不是好主意。这样做, 会把部分构造的对象暴露给新的线程。如果对象拥有一个线程, 那么它应该提供一个启动该线程的 `start()` 或 `init()` 方法, 而不是从构造器中启动它。

3.2: 结束线程

线程会以以下三种方式之一结束:

- 线程到达其 `run()` 方法的末尾。
- 线程抛出一个未捕获到的 `Exception` 或 `Error`。
- 另一个线程调用一个弃用的 `stop()` 方法。弃用是指这些方法仍然存在, 但是您不应该在新代码中使用它们, 并且应该尽量从现有代码中除去它们。

当 Java 程序中的所有线程都完成时, 程序就退出了。

3.3: 加入线程

`Thread` API 包含了等待另一个线程完成的方法: `join()` 方法。当调用 `Thread.join()` 时, 调用线程将阻塞, 直到目标线程完成为止。

`Thread.join()` 通常由使用线程的程序使用, 以将大问题划分成许多小问题, 每个小问题分配一个线程。本章结尾处的示例创建了十个线程, 启动它们, 然后使用 `Thread.join()` 等待它们全部完成。

3.4: 调度

除了何时使用 `Thread.join()` 和 `Object.wait()` 外, 线程调度和执行的计时是不确定的。如果两个线程同时运行, 而且都不等待, 您必须假设在任何两个指令之间, 其它线程都可以运行并修改程序变量。如果线程要访问其它线程可以看见的变量, 如从静态字段(全局变量)直接或间接引用的数据, 则必须使用同步以确保数据一致性。

在以下的简单示例中, 我们将创建并启动两个线程, 每个线程都打印两行到 `System.out`:

```
public class TwoThreads {

    public static class Thread1 extends Thread {
        public void run() {
            System.out.println("A");
            System.out.println("B");
        }
    }

    public static class Thread2 extends Thread {
        public void run() {
            System.out.println("1");
            System.out.println("2");
        }
    }

    public static void main(String[] args) {
        new Thread1().start();
        new Thread2().start();
    }
}
```

我们并不知道这些行按什么顺序执行,只知道“1”在“2”之前打印,以及“A”在“B”之前打印。输出可能是以下结果中的任何一种:

- 1 2 A B
- 1 A 2 B
- 1 A B 2
- A 1 2 B
- A 1 B 2
- A B 1 2

不仅不同机器之间的结果可能不同,而且在同一机器上多次运行同一程序也可能生成不同结果。永远不要假设一个线程会在另一个线程之前执行某些操作,除非您已经使用了同步以强制一个特定的执行顺序。

3.5: 休眠

Thread API 包含了一个 `sleep()` 方法,它将使当前线程进入等待状态,直到过了一段指定时间,或者直到另一个线程对当前线程的 Thread 对象调用了 `Thread.interrupt()`,从而中断了线程。当过了指定时间后,线程又将变成可运行的,并且回到调度程序的可运行线程队列中。

如果线程是由对 `Thread.interrupt()` 的调用而中断的,那么休眠的线程会抛出 `InterruptedException`,这样线程就知道它是由中断唤醒的,就不必查看计时器是否过期。

`Thread.yield()` 方法就象 `Thread.sleep()` 一样,但它并不引起休眠,而只是暂停当前线程片刻,这样其它线程就可以运行了。在大多数实现中,当较高优先级的线程调用 `Thread.yield()` 时,较低优先级的线程就不会运行。

`CalculatePrimes` 示例使用了一个后台线程计算素数,然后休眠十秒钟。当计时器过期后,它就会设置一个标志,表示已经过了十秒。

3.6: 守护程序线程

我们提到过当 Java 程序的所有线程都完成时,该程序就退出,但这并不完全正确。隐藏的系统线程,如垃圾收集线程和由 JVM 创建的其它线程会怎么样?我们没有办法停止这些线程。如果那些线程正在运行,那么 Java 程序怎么退出呢?

这些系统线程称作守护程序线程。Java 程序实际上是在它的所有非守护程序线程完成后退出的。

任何线程都可以变成守护程序线程。

三：多线程的同步

1：临界资源问题

多个线程间共享的数据称为共享资源或临界资源，由于是线程调度器负责线程的调度，程序员无法精确控制多线程的交替顺序。这种情况下，多线程对临界资源的访问有时会导致数据的不一致性。

这个问题的产生可以看下面的例子：

想象一个表示栈的类。这个类最初可能象下面那样：

```
1. public class MyStack {
2.
3.     int idx = 0;
4.     char [] data = new char[6];
5.
6.     public void push(char c) {
7.         data[idx] = c;
8.         idx++;
9.     }
10.
11.    public char pop() {
12.        idx--;
13.        return data[idx];
14.    }
15. }
```

注意这个类没有处理栈的上溢和下溢，所以栈的容量是相当有限的。这些方面和本讨论无关。

这个模型的行为要求索引值包含栈中下一个空单元的数组下标。“先进后出”方法用来产生这个信息。

现在想象两个线程都有对这个类里的一个单一实例的引用。一个线程将数据推入栈，而另一个线程，或多或少独立地，将数据弹出栈。通常看来，数据将会正确地被加入或移走。然而，这存在着潜在的问题。

假设线程 a 正在添加字符，而线程 b 正在移走字符。线程 a 已经放入了一个字符，但还没有使下标加 1。因为某个原因，这个线程被剥夺(运行的机会)。这时，对象所表示的数据模型是不一致的。

```
buffer |p|q|r| | | |
idx = 2 ^
```

特别地，一致性会要求 idx=3，或者还没有添加字符。

如果线程 a 恢复运行，那就可能不造成破坏，但假设线程 b 正等待移走一个字符。在线程 a 等待另一个运行的机会时，线程 b 正在等待移走一个字符的机会。

pop() 方法所指向的条目存在不一致的数据，然而 pop 方法要将下标值减 1。

```
buffer |p|q|r| | | |
idx = 1 ^
```

这实际上将忽略了字符“r”。此后，它将返回字符“q”。至此，从其行为来看，就好像没有推入字母“r”，所以很难说是否存在问题。现在看一看如果线程 a 继续运行，会发生什么。

线程 a 从上次中断的地方开始运行，即在 push() 方法中，它将使下标值加 1。现在你可以看到：

```
buffer |p|q|r| | | |  
idx = 2 ^
```

注意这个配置隐含了：“q”是有效的，而含有“r”的单元是下一个空单元。也就是说，读取“q”时，它就像被两次推入了栈，而字母“r”则永远不会出现。

这是一个当多线程共享数据时会经常发生的问题的一个简单范例。需要有机制来保证共享数据在任何线程使用它完成某一特定任务之前是一致的。

2 互斥锁

在 Java 语言中，为保证共享数据操作的完整性，引入了对象互斥锁的概念。每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。这好似学校里专用教室的使用规则：为保证高年级学生顺利进行课程设计，将个别教室规定为“专教”并在门上标明。任意时刻，只能有一个班级使用该教室，直至他们课程设计活动结束。即使中午或周末休息，也不必担心教室被其他的班级占用，或黑板上的文字被他人擦除。显然，这种做法会降低教室的利用率，因此除特别标明以外，教室应该是公用的，哪个班级都可以使用，由教务处负责安排调度。

Java 对象默认也是可以被多线程共用的，在需要时才启动“互斥”机制，成为专用对象。关键字 synchronized 来与对象的互斥锁联系。当某个对象用 synchronized 修饰时，表明该对象已启动“互斥”机制，在任一时刻只能由一个线程访问。即使该线程出现阻塞，该对象的被锁定状态也不会解除，其他线程仍不能访问该对象。

synchronized 关键字的使用方式有两种：

- I 用在对象前面限制一段代码的执行
- I 用在方法声明中，表示整个方法为同步方法。

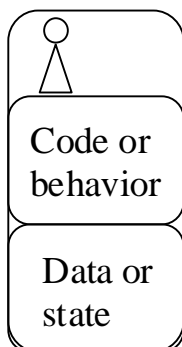
看一看下面修改过的代码片断：

```
public void push(char c) {  
    synchronized(this) {  
        data[idx] = c;  
        idx++;  
    }  
}
```

当线程运行到 synchronized 语句，它检查作为参数传递的对象，并在继续执行之前试图从对象获得锁标志。

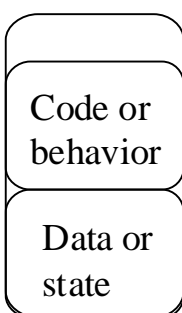
2.1: 对象锁标志

Object this

**Thread before** synchronized(this)

```
public void push(char c) {
    synchronized (this) {
        data[idx] = c;
        idx++;
    }
}
```

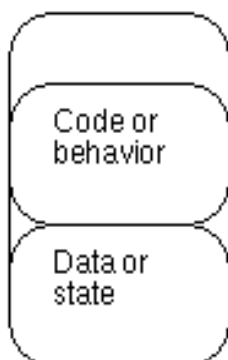
Object this

**Thread after** synchronized(this)

```
public void push(char c) {
    synchronized (this) {
        data[idx] = c;
        idx++;
    }
}
```

意识到它自身并没有保护数据是很重要的。因为如果同一个对象的 pop() 方法没有受到 synchronized 的影响，且 pop() 是由另一个线程调用的，那么仍然存在破坏 data 的一致性的危险。如果要使锁有效，所有存取共享数据的方法必须在同一把锁上同步。

下图显示了如果 pop() 受到 synchronized 的影响，且另一个线程在原线程持有那个对象的锁时试图执行 pop() 方法时所发生的事情：

Object this
Lock flag missingThread, trying to execute
synchronized(this)

```
Waiting for object lock | public char pop() {
                          |     synchronized (this) {
                          |         idx--;
                          |         return data[idx];
                          |     }
                          | }
```

当线程试图执行 `synchronized(this)` 语句时, 它试图从 `this` 对象获取锁标志。由于得不到标志, 所以线程不能继续运行。然后, 线程加入到与那个对象锁相关联的等待线程池中。当标志返回给对象时, 某个等待这个标志的线程将得到这把锁并继续运行。

2.2 释放锁标志

由于等待一个对象的锁标志的线程在得到标志之前不能恢复运行, 所以让持有锁标志的线程在不再需要的时候返回标志是很重要的。

锁标志将自动返回给它的对象。持有锁标志的线程执行到 `synchronized()` 代码块末尾时将释放锁。Java 技术特别注意了保证即使出现中断或异常而使得执行流跳出 `synchronized()` 代码块, 锁也会自动返回。此外, 如果一个线程对同一个对象两次发出 `synchronized` 调用, 则在跳出最外层的块时, 标志会正确地释放, 而最内层的将被忽略。

这些规则使得与其它系统中的等价功能相比, 管理同步块的使用简单了很多。

2.3 `synchronized`——放在一起

`synchronized`——放在一起

- ! 所有对易碎数据的存取应当同步。
- ! 由 `synchronized` 保护的易碎数据应当是 `private` 的。

正如所暗示的那样, 只有当所有对易碎数据的存取位于同步块内, `synchronized()` 才会发生作用。

所有由 `synchronized` 块保护的易碎数据应当标记为 `private`。考虑来自对象的易碎部分的数据的可存取性。如果它们不被标记为 `private`, 则它们可以由位于类定义之外的代码存取。这样, 你必须确信其他程序员不会省略必需的保护。

一个方法, 如果它全部属于与这个实例同步的块, 它可以把 `synchronized` 关键字放到它的头部。下面两段代码是等价的:

```
public void push(char c) {
    synchronized(this) {
        :
        :
    }
}

public synchronized void push(char c) {
    :
    :
}
```

为什么使用另外一种技术?

如果你把 `synchronized` 作为一种修饰符, 那么整个块就成为一个同步块。这可能会导致不必要地持有锁标志很长时间, 因而是低效的。

然而, 以这种方式来标记方法可以使方法的用户由 `javadoc` 产生的文档了解到: 正在同步。这对于设计时避免死锁(将在下一节讨论)是很重要的。注意 `javadoc` 文档生成器将 `synchronized` 关键字传播到文档文件中, 但它不能为在方法块内的 `synchronized(this)` 做到这点。

3 线程的死锁

如果程序中有多个线程竞争多个资源,就可能会产生死锁。当一个线程等待由另一个线程持有的锁,而后者正在等待已被第一个线程持有的锁时,就会发生死锁。在这种情况下,除非另一个已经执行到 `synchronized` 块的末尾,否则没有一个线程能继续执行。由于没有一个线程能继续执行,所以没有一个线程能执行到块的末尾。

Java 技术不监测也不试图避免这种情况。因而保证不发生死锁就成了程序员的责任。避免死锁的一个通用的经验法则是:决定获取锁的次序并始终遵照这个次序。按照与获取相反的次序释放锁。

讨论下面的情况,两个线程 A、B 用到同一个对象 s (s 为共享资源),且线程 A 在执行中要用到 B 运行后所创造的条件(例如,使用 `B.join()` 语句与 B 建立同步)。在这种前提下 A 先开始运行,进入同步语句块后,对象 s 被锁定,接着线程 A 因等待 B 运行结束这一条件而进入阻塞状态。于是线程 B 开始运行,但因无法访问对象 s (已在 A 中被锁定),线程 B 也进入阻塞状态,等待 s 被线程 A 解除锁定。最终的结果是,两个线程互相等待,都无法运行,这种状态称为线程的死锁。

要避免死锁,应该确保在获取多个锁时,在所有的线程中都以相同的顺序获取锁。

4 wait()和 notify()方法

使用 `synchronized` 关键字锁定 Java 对象,就好似规定某个教室为专用教室一样,不可避免的会降低程序效率。

为了实现线程阻塞时释放其锁定的共享资源,以给其他的线程提供运行的机会。`java.lang` 包中定义了几个有用的方法: `wait()`, `notify()`, `notifyAll()`。

在同步方法(语句块)中,被锁定的对象可以调用 `wait()` 方法,这将导致当前线程被阻塞并放弃该对象的互斥锁,即解除了 `wait()` 方法的当前对象的锁定状态,其他的线程就有机会访问该对象。因调用了 `wait()` 方法而阻塞的线程,将被加入一个特殊的对象等待队列中,直到调用该 `wait()` 方法的对象在其他的线程中调用了 `notify()` 或 `notifyAll()` 方法,这种等待才可能解除。

如果有多个线程因调用了 `wait()` 方法而在等待同一个对象,则应使用该对象在其他线程中调用 `notifyAll()` 方法,以使等待队列上的所有线程离开阻塞状态。而 `notify()` 方法每次运行只能唤醒等待队列中一个线程,至于是哪一个被唤醒则由线程调度器决定,程序员是无法控制的。引入了 `wait()`/`notify()` 方法后线程的状态转换情况如图所示:

经常创建不同的线程来执行不相关的任务。然而,有时它们所执行的任务是有某种联系的,为此必须编写使它们交互的程序。

4.1 场景

把你自己和出租车司机当作两个线程。你需要出租车司机带你到终点,而出租车司机需

要为乘客服务来获得车费。所以,你们两者都有一个任务。

4.2 问题

你希望坐到出租车里, 舒服地休息, 直到出租车司机告诉你已经到达终点。如果每 2 秒就问一下“我们到了哪里?”, 这对出租车司机和你都会是很烦的。出租车司机想睡在出租车里, 直到一个乘客想到另外一个地方去。出租车司机不想为了查看是否有乘客的到来而每 5 分钟就醒来一次。所以, 两个线程都想用一种尽量轻松的方式来达到它们的目的。

4.3 解决方案

出租车司机和你都想用某种方式进行通信。当你正忙着走向出租车站时, 司机正在车中安睡。当你告诉司机你想坐他的车时, 司机醒来并开始驾驶, 然后你开始等待并休息。到达终点时, 司机会通知你, 所以你必须继续你的任务, 即走出出租车, 然后去工作。出租车司机又开始等待和休息, 直到下一个乘客的到来。

4.4 wait()和 notify()

java.lang.Object 类中提供了两个用于线程通信的方法: wait()和 notify()。如果线程对一个同步对象 x 发出一个 wait()调用, 该线程会暂停执行, 直到另一个线程对同一个同步对象 x 也发出一个 wait()调用。

在上个场景中, 在车中等待的出租车司机被翻译成执行 cab.wait()调用的“出租车司机”线程, 而你使用出租车的需求被翻译成执行 cab.notify()调用的“你”线程。

为了让线程对一个对象调用 wait()或 notify(), 线程必须锁定那个特定的对象。也就是说, 只能在它们被调用的实例的同步块内使用 wait()和 notify()。对于这个实例来说, 需要一个以 synchronized(cab)开始的块来允许执行 cab.wait()和 cab.notify()调用。

关于池

当线程执行包含对一个特定对象执行 wait()调用的同步代码时, 那个线程被放到与那个对象相关的等待池中。此外, 调用 wait()的线程自动释放对象的锁标志。可以调用不同的 wait():

wait() 或 wait(long timeout);

对一个特定对象执行 notify()调用时, 将从对象的等待池中移走一个任意的线程, 并放到锁池中, 那里的对象一直在等待, 直到可以获得对象的锁标记。notifyAll()方法将从等待池中移走所有等待那个对象的线程并放到锁池中。只有锁池中的线程能获取对象的锁标记, 锁标记允许线程从上次因调用 wait()而中断的地方开始继续运行。

在许多实现了 wait()/notify()机制的系统中, 醒来的线程必定是那个等待时间最长的线程。然而, 在 Java 技术中, 并不保证这点。

注意, 不管是否有线程在等待, 都可以调用 notify()。如果对一个对象调用 notify()方法, 而在这个对象的锁标记等待池中并没有阻塞的线程, 那么 notify()调用将不起任何作用。对 notify()的调用不会被存储。

4.5 同步的监视模型

同步的监视模型

- I 使共享数据处于一致的状态
- I 保证程序不死锁
- I 不要将期待不同通知的线程放到同一个等待

协调两个需要存取公共数据的线程可能会变得非常复杂。你必须非常小心,以保证可能有另一个线程存取数据时,共享数据的状态是一致的。因为线程不能在其他线程在等待这把锁的时候释放合适的锁,所以你必须保证你的程序不发生死锁,

在出租车范例中,代码依赖于一个同步对象——出租车,在其上执行 `wait()` 和 `notify()`。如果有任何人在等待一辆公共汽车,你就需要一个独立的公共汽车对象,在它上面施用 `notify()`。记住,在同一个等待池中的所有线程都因来自等待池的控制对象的通知而满足。永远不要设计这样的程序:把线程放在同一个等待池中,但它们却在等待不同条件的通知。

4.6 放在一起

下面将给出一个线程交互的实例,它说明了如何使用 `wait()` 和 `notify()` 方法来解决一个经典的生产者—消费者问题。

我们先看一下栈对象的大致情况和要存取栈的线程的细节。然后再看一下栈的详情,以及基于栈的状态来保护栈数据和实现线程通信的机制。

实例中的栈类称为 `SyncStack`,用来与核心 `java.util.Stack` 相区别,它提供了如下公共的 API:

```
public synchronized void push(char c);
public synchronized char pop();
```

生产者线程运行如下方法:

```
public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
        c = (char)(Math.random() * 26 + 'A');
        theStack.push(c);
        System.out.println("Producer" + num + ": " + c);
        try {
            Thread.sleep((int)(Math.random() * 300));
        } catch (InterruptedException e) {
            // ignore it
        }
    }
}
```

这将产生 200 个随机的大写字母并将其推入栈中,每个推入操作之间有 0 到 300 毫秒的随机延迟。每个被推入的字符将显示到控制台上,同时还显示正在执行的生产者线程的标识。

消费者

消费者线程运行如下方法:

```
public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
        c = theStack.pop();
        System.out.println(" Consumer" + num + ": " + c);
        try {
            Thread.sleep((int)(Math.random() * 300));
        }
    }
}
```

```
    } catch (InterruptedException e) {  
        // ignore it  
    }  
}  
}
```

上面这个程序从栈中取出 200 个字符,每两个取出操作的尝试之间有 0 到 300 毫秒的随机延迟。每个被弹出的字符将显示在控制台上,同时还显示正在执行的消费者线程的标识。

现在考虑栈类的构造。你将使用 `Vector` 类创建一个栈,它看上去有无限大的空间。按照这种设计,你的线程只要在栈是否为空的基础上进行通信即可。

SyncStack 类

一个新构造的 `SyncStack` 对象的缓冲应当为空。下面这段代码用来构造你的类:

```
public class SyncStack {  
    private Vector buffer = new Vector(400,200);  
    public synchronized char pop() {  
    }  
    public synchronized void push(char c) {  
    }  
}
```

请注意,其中没有任何构造函数。包含有一个构造函数是一种相当好的风格,但为了保持简洁,这里省略了构造函数。

现在考虑 `push()` 和 `pop()` 方法。为了保护共享缓冲,它们必须均为 `synchronized`。此外,如果要执行 `pop()` 方法时栈为空,则正在执行的线程必须等待。若执行 `push()` 方法后栈不再为空,正在等待的线程将会得到通知。

`pop()` 方法如下:

```
public synchronized char pop() {  
    char c;  
    while (buffer.size() == 0) {  
        try {  
            this.wait();  
        } catch (InterruptedException e) {  
            // ignore it  
        }  
    }  
    c = ((Character)buffer.remove(buffer.size()-1)).charValue();  
    return c;  
}
```

注意这里显式地调用了栈对象的 `wait()`,这说明了如何对一个特定对象进行同步。如果栈为空,则不会弹出任何数据,所以一个线程必须等到栈不再为空时才能弹出数据。

由于一个 `interrupt()` 的调用可能结束线程的等待阶段,所以 `wait()` 调用被放在一个 `try/catch` 块中。对于本例,`wait()` 还必须放在一个循环中。如果 `wait()` 被中断,而栈仍为

空, 则线程必须继续等待。

栈的 `pop()` 方法为 `synchronized` 是出于两个原因。首先, 将字符从栈中弹出影响了共享数据 `buffer`。其次, `this.wait()` 的调用必须位于关于栈对象的一个同步块中, 这个块由 `this` 表示。

你将看到 `push()` 方法如何使用 `this.notify()` 方法将一个线程从栈对象的等待池中释放出来。一旦线程被释放并可随后再次获得栈的锁, 该线程就可以继续执行 `pop()` 完成从栈缓冲区中移走字符任务的代码。

注 — 在 `pop()` 中, `wait()` 方法在对栈的共享数据作修改之前被调用。这是非常关键的一点, 因为在对象锁被释放和线程继续执行改变栈数据的代码之前, 数据必须保持一致的状态。你必须使你所设计的代码满足这样的假设: 在进入影响数据的代码时, 共享数据是处于一致的状态。

需要考虑的另一点是错误检查。你可能已经注意到没有显式的代码来保证栈不发生下溢。这不是必需的, 因为从栈中移走字符的唯一方法是通过 `pop()` 方法, 而这个方法导致正在执行的线程在没有字符的时候会进入 `wait()` 状态。因此, 错误检查不是必要的。`push()` 在影响共享缓冲方面与此类似, 因此也必须被同步。此外, 由于 `push()` 将一个字符加入缓冲区, 所以由它负责通知正在等待非空栈的线程。这个通知的完成与栈对象有关。

`push()` 方法如下:

```
public synchronized void push(char c) {
    this.notify();
    Character charObj = new Character(c);
    buffer.addElement(charObj);
}
```

对 `this.notify()` 的调用将释放一个因栈空而调用 `wait()` 的单个线程。在共享数据发生真正的改变之前调用 `notify()` 不会产生任何结果。只有退出该 `synchronized` 块后, 才会释放对象的锁, 所以当栈数据在被改变时, 正在等待锁的线程不会获得这个锁。

4.7 SyncStack 范例

完整的代码

现在, 生产者、消费者和栈代码必须组装成一个完整的类。还需要一个测试工具将这些代码集成为一体。特别要注意, `SyncTest` 是如何只创建一个由所有线程共享的栈对象的。

`SyncTest.java`

```
1.package mod14;
2.public class SyncTest {
3.public static void main(String args[]) {
4.
5.SyncStack stack = new SyncStack();
6.
7.Producer p1 = new Producer(stack);
8.Thread prodT1= new Thread(p1);
9.prodT1.start();
```

```
10.
11.Producer p2 = new Producer(stack);
12.Thread prodT2= new Thread(p2);
13.prodT2.start();
14.
15.Consumer c1 = new Consumer(stack);
16.Thread consT1 = new Thread(c1);
17.consT1.start();
18.
19.Consumer c2 = new Consumer(stack);
20.Thread consT2 = new Thread(c2);
21.constT2.start();
22.}
23.}
```

Producer.java

```
1.package mod14;
2.public class Producer implements Runnable {
3.private SyncStack theStack;
4.private int num;
5.private static int counter = 1;
6.public Producer (SyncStack s) {
7.theStack = s;
8.num = counter++;
9.}
10.
11.public void run() {
12.char c;
13.
14.for (int i = 0; i < 200; i++) {
15.c = (char)(Math.random() * 26 + `A');
16.theStack.push(c);
17.System.out.println("Producer" + num + ": " + c);
18.try {
19.Thread.sleep((int)(Math.random() * 300));
20.} catch (InterruptedException e) {
21.// ignore it
22.}
23.}
24.}
25.}
```

Consumer.java

```
1.package mod14;
2.public class Consumer implements Runnable {
3.private SyncStack theStack;
4.private int num;
5.private static int counter = 1;
6.
7.public Consumer (SyncStack s) {
8.theStack = s;
9.num = counter++;
10.}
11.
12.public void run() {
13.char c;
14.for (int i=0; i < 200; i++) {
15.c = theStack.pop();
16.System.out.println("Consumer" + num + ": " + c);
17.try {
18.Thread.sleep((int)(Math.random() * 300));
19.} catch (InterruptedException e) {
20.// ignore it
21.}
22.}
23.}
24.}
```

SyncStack.java

```
1.package mod14;
2.
3.import java.util.Vector;
4.
5.public class SyncStack {
6.private Vector buffer = new Vector(400,200);
7.
8.public synchronized char pop() {
9.char c;
10.
11.while (buffer.size() == 0) {
12.try {
13.this.wait();
14.} catch (InterruptedException e) {
15.// ignore it
16.}
17.}
18.}
```

```
19. c = ((Character)buffer.remove(buffer.size()- 1).charValue());
20. return c;
21. }
22. public synchronized void push(char c) {
23. this.notify();
24. Character charObj = new Character(c);
25. buffer.addelement(charObj);
26. }
27. }
```

运行 javamodb.SyncTest 的输出如下。请注意每次运行线程代码时,结果都会有所不同。

```
Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
Consumer1: F
Producer2: M
Consumer2: M
Consumer2: T
```

Java 语言包含了内置在语言中的功能强大的线程工具。您可以将线程工具用于:

- 增加 GUI 应用程序的响应速度
- 利用多处理器系统
- 当程序有多个独立实体时,简化程序逻辑
- 在不阻塞整个程序的情况下,执行阻塞 I/O

当使用多个线程时,必须谨慎,遵循在线程之间共享数据的规则,我们将在共享对数据的访问中讨论这些规则。所有这些规则归结为一条基本原则: **不要忘了同步。**

5: 同步的应用

5.1 可见性同步

跨线程维护正确的可见性, 只要在几个线程之间共享非 `final` 变量, 就必须使用 `synchronized` 以确保一个线程可以看见另一个线程做的更改。

可见性同步的基本规则是在以下情况中必须同步:

- 读取上一次可能是由另一个线程写入的变量
- 写入下一次可能由另一个线程读取的变量

5.2 用于一致性的同步

除了用于可见性的同步, 从应用程序角度看, 您还必须用同步来确保一致性得到了维护。当修改多个相关值时, 您想要其它线程原子地看到这组更改 — 要么看到全部更改, 要么什么也看不到。这适用于相关数据项 (如粒子的位置和速率) 和元数据项 (如链表中包含的数据值和列表自身中的数据项的链)。

考虑以下示例, 它实现了一个简单 (但不是线程安全的) 的整数堆栈:

```
public class UnsafeStack {  
    public int top = 0;  
    public int[] values = new int[1000];  
    public void push(int n) {  
        values[top++] = n;  
    }  
    public int pop() {  
        return values[--top];  
    }  
}
```

如果多个线程试图同时使用这个类, 会发生什么? 这可能是个灾难。因为没有同步, 多个线程可以同时执行 `push()` 和 `pop()`。如果一个线程调用 `push()`, 而另一个线程正好在递增了 `top` 并要把它用作 `values` 的下标之间调用 `push()`, 会发生什么? 结果, 这两个线程会把它们的新值存储到相同的位置! 当多个线程依赖于数据值之间的已知关系, 但没有确保只有一个线程可以在给定时间操作那些值时, 可能会发生许多形式的数据损坏, 而这只是其中之一。

对于这种情况, 补救办法很简单: 同步 `push()` 和 `pop()` 这两者, 您将防止线程执行相互干扰。请注意, 需要使用 `synchronized` 来确保 `top` 和 `values` 之间的关系保持一致。

5.3 不变性和 final 字段

许多 Java 类, 包括 String、Integer 和 BigDecimal, 都是不可改变的: 一旦构造之后, 它们的状态就永远不会更改。如果某个类的所有字段都被声明成 final, 那么这个类就是不可改变的。(实际上, 许多不可改变的类都有非 final 字段, 用于高速缓存以前计算的方法结果, 如 String.hashCode(), 但调用者看不到这些字段。)

不可改变的类使并发编程变得非常简单。因为不能更改它们的字段, 所以就不需要担心把状态的更改从一个线程传递到另一个线程。在正确构造了对象之后, 可以把它看作是常量。

同样, final 字段对于线程也更友好。因为 final 字段在初始化之后, 它们的值就不能更改, 所以当在线程之间共享 final 字段时, 不需要担心同步访问

5.4 什么时候不需要同步

在某些情况中, 您不必用同步来将数据从一个线程传递到另一个, 因为 JVM 已经隐含地为您执行同步。这些情况包括:

- 由静态初始化器 (在静态字段上或 static{} 块中的初始化器) 初始化数据时
- 访问 final 字段时
- 在创建线程之前创建对象时
- 线程可以看见它将要处理的对象时

5.5 性能考虑事项

关于同步的性能代价有许多说法 — 其中有许多是错的。同步, 尤其是争用的同步, 确实有性能问题, 但这些问题并没有象人们普遍怀疑的那么大。许多人都使用别出心裁但不起作用的技巧以试图避免必须使用同步, 但最终都陷入了麻烦。一个典型的示例是双重检查锁定模式。这种看似无害的结构据说可以避免公共代码路径上的同步, 但却令人费解地失败了, 而且所有试图修正它的尝试也失败了。

在编写并发代码时, 除非看到性能问题的确凿证据, 否则不要过多考虑性能。瓶颈往往出现在我们最不会怀疑的地方。投机性地优化一个也许最终根本不会成为性能问题的代码路径 — 以程序正确性为代价 — 是一桩赔本的生意。

5.6 同步准则

当编写 synchronized 块时, 有几个简单的准则可以遵循, 这些准则在避免死锁和性能危险的风险方面大有帮助:

使代码块保持简短。Synchronized 块应该简短 — 在保证相关数据操作的完整性的同时, 尽量简短。把不随线程变化的预处理和后处理移出 synchronized 块。

不要阻塞。不要在 synchronized 块或方法中调用可能引起阻塞的方法, 如 InputStream.read()。

在持有锁的时候,不要对其它对象调用方法。这听起来可能有些极端,但它消除了最常见的死锁源头。

6: 线程有关的 API

除了使用轮询(它可能消耗大量 CPU 资源,而且具有计时不精确的特征),Object 类还包括一些方法,可以让线程相互通知事件的发生。Object 类定义了 `wait()`、`notify()` 和 `notifyAll()` 方法。要执行这些方法,必须拥有相关对象的锁。`wait()` 会让调用线程休眠,直到用 `Thread.interrupt()` 中断它、过了指定的时间、或者另一个线程用 `notify()` 或 `notifyAll()` 唤醒它。当对某个对象调用 `notify()` 时,如果有任何线程正在通过 `wait()` 等待该对象,那么就会唤醒其中一个线程。当对某个对象调用 `notifyAll()` 时,会唤醒所有正在等待该对象的线程。

这些方法是更复杂的锁定、排队和并发性代码的构件。但是,`notify()` 和 `notifyAll()` 的使用很复杂。尤其是,使用 `notify()` 来代替 `notifyAll()` 是有风险的。除非您确实知道正在做什么,否则就使用 `notifyAll()`。

与其使用 `wait()` 和 `notify()` 来编写您自己的调度程序、线程池、队列和锁,倒不如使用 `util.concurrent` 包,这是一个被广泛使用的开放源码工具箱,里面都是有用的并发性实用程序。JDK 1.5 将包括 `java.util.concurrent` 包;它的许多类都派生自 `util.concurrent`。

6.1: 线程优先级

Thread API 让您可以将执行优先级与每个线程关联起来。但是,这些优先级如何映射到底层操作系统调度程序取决于实现。在某些实现中,多个 — 甚至全部 — 优先级可能被映射成相同的底层操作系统优先级。

在遇到诸如死锁、资源匮乏或其它意外的调度特征问题时,许多人都想要调整线程优先级。但是,通常这样只会把问题移到别的地方。大多数程序应该完全避免更改线程优先级。

6.2: 线程组

ThreadGroup 类原本旨在用于把线程集合构造成组。但是,结果证明 ThreadGroup 并没有那样有用。您最好只使用 Thread 中的等价方法。

ThreadGroup 确实提供了一个有用的功能部件(Thread 中目前还没有): `uncaughtException()` 方法。线程组中的某个线程由于抛出了未捕获的异常而退出时,会调用 `ThreadGroup.uncaughtException()` 方法。这就让您有机会关闭系统、将一条消息写到日志文件或者重新启动失败的服务

6.3: SwingUtilities

虽然 `SwingUtilities` 类不是 Thread API 的一部分,但还是值得简单提一下。

正如前面提到的,Swing 应用程序有一个 UI 线程(有时称为事件线程),所有 UI 活动都必须在这个线程中发生。有时,另一个线程也许想要更新屏幕上某样东西的外观,或者触发 Swing 对象上的一个事件。

`SwingUtilities.invokeLater()` 方法可以让您将 `Runnable` 对象传送给它,并且在事件线程中执行指定的 `Runnable`。它的同类 `invokeAndWait()` 会在事件线程中调用

Runnable, 但 `invokeAndWait()` 会阻塞, 直到 Runnable 完成执行之后。

```
void showHelloThereDialog() throws Exception {  
    Runnable showModalDialog = new Runnable() {  
        public void run() {  
            JOptionPane.showMessageDialog(myMainFrame, "Hello There");  
        }  
    };  
    SwingUtilities.invokeLater(showModalDialog);  
}
```

对于 AWT 应用程序, `java.awt.EventQueue` 还提供了 `invokeLater()` 和 `invokeAndWait()`。

6.4: `suspend()`和`resume()`方法

- | |
|--|
| <p style="text-align: center;"><code>suspend()</code>和<code>resume()</code>方法</p> <ul style="list-style-type: none">! JDK1.2 不赞成使用它们! 应当用 <code>wait()</code>和<code>notify()</code>来代替它们 |
|--|

JDK1.2 中不赞成使用 `suspend()`和`resume()`方法。`resume()`方法的唯一作用就是恢复被挂起的线程。所以, 如果没有 `suspend()`, `resume()`也就没有存在的必要。从设计的角度来看, 有两个原因使 `suspend()`非常危险: 它容易产生死锁; 它允许一个线程控制另一个线程代码的执行。下面将分别介绍这两种危险。

假设有两个线程: `threadA` 和 `threadB`。当正在执行它的代码时, `threadB` 获得一个对象的锁, 然后继续它的任务。现在 `threadA` 的执行代码调用 `threadB.suspend()`, 这将使 `threadB` 停止执行它的代码。

如果 `threadB.suspend()`没有使 `threadB` 释放它所持有的锁, 就会发生死锁。如果调用 `threadB.resume()`的线程需要 `threadB` 仍持有的锁, 这两个线程就会陷入死锁。

假设 `threadA` 调用 `threadB.suspend()`。如果 `threadB` 被挂起时 `threadA` 获得控制, 那么 `threadB` 就永远得不到机会来进行清除工作, 例如使它正在操作的共享数据处于稳定状态。为了安全起见, 只有 `threadB` 才可以决定何时停止它自己的代码。

你应该使用对同步对象调用 `wait()`和`notify()`的机制来代替 `suspend()`和`resume()`进行线程控制。这种方法是通过执行 `wait()`调用来强制线程决定何时“挂起”自己。这使得同步对象的锁被自动释放, 并给予线程一个在调用 `wait()`之前稳定任何数据的机会。

6.5: `stop()`方法

- | |
|---|
| <p style="text-align: center;"><code>stop()</code>方法</p> <ul style="list-style-type: none">! 在终止前释放锁。! 可能使共享数据处于不一致的状态。! 应当用 <code>wait()</code>和<code>notify()</code>来代替它们 |
|---|

`stop()` 方法的情形是类似的, 但结果有所不同。如果一个线程在持有一个对象锁的时候被停止, 它将在终止之前释放它持有的锁。这避免了前面所讨论的死锁问题, 但它又引入了其他问题。

在前面的范例中, 如果线程在已将字符加入栈但还没有使下标值加 1 之后被停止, 你在释放锁的时候会得到一个不一致的栈结构。

总会有一些关键操作需要不可分割地执行, 而且在线程执行这些操作时被停止就会破坏操作的不可分割性。

一个关于停止线程的独立而又重要的问题涉及线程的总体设计策略。创建线程来执行某个特定作业, 并存活于整个程序的生命周期。换言之, 你不会这样来设计程序: 随意地创建和处理线程, 或创建无数个对话框或 `socket` 端点。每个线程都会消耗系统资源, 而系统资源并不是无限的。这并不是暗示一个线程必须连续执行; 它只是简单地意味着应当使用合适而安全的 `wait()` 和 `notify()` 机制来控制线程。

练习实践

本章的内容为多线程，实践重点：

I 多线程的含义与应用

程序 1

多线程基础

需求：建立两个线程，各显示 0~500 的数。

目标：

- 1、线程的概念；
- 2、线程的创建；
- 3、理解线程与进程的区别；
- 4、线程的使用。

程序：

```
package com.useful.java.part5;

public class TestThread extends Thread {

    int intNumberThread = 0;
    int intNumberThreadMain = 0;
    int intTotal = 500;

    public TestThread() {
    }

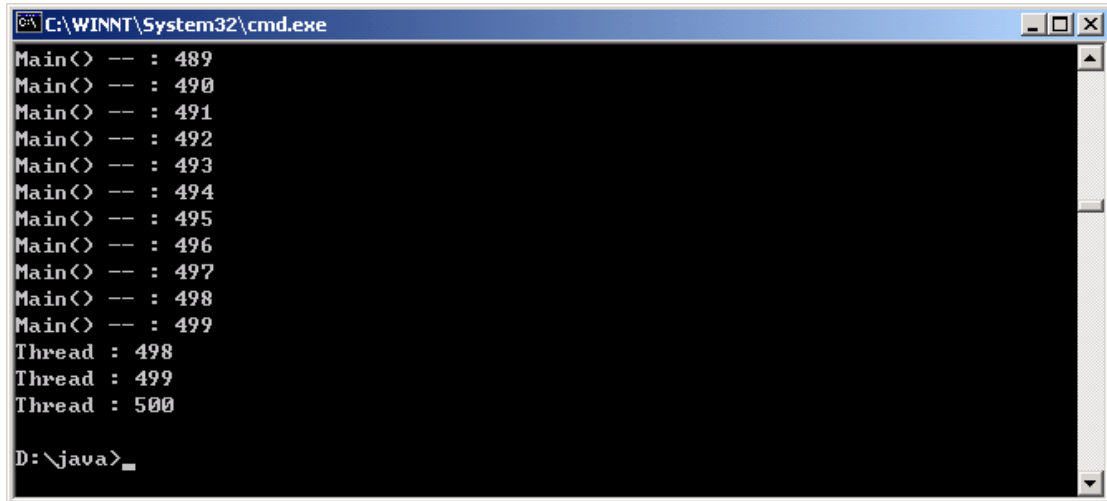
    public void run(){
        while(true){
            System.out.println("Thread : " + intNumberThread);
            intNumberThread++;
            if(intNumberThread > intTotal){
                return;
            }
        }
    }

    public static void main(String[] args) {
        TestThread testThread1 = new TestThread();
        testThread1.start();
        while(testThread1.intNumberThreadMain < testThread1.intTotal){
            System.out.println("Main()      --      :      "      +
testThread1.intNumberThreadMain);
            testThread1.intNumberThreadMain++;
        }
    }
}
```

```
        }  
    }  
}
```

说明:

- 1、创建的常用方法之一就是继承 Thread, 然后覆盖 run();
- 2、线程的执行无先后之分;
- 3、运行情况如下图:



```
C:\WINNT\System32\cmd.exe  
Main() -- : 489  
Main() -- : 490  
Main() -- : 491  
Main() -- : 492  
Main() -- : 493  
Main() -- : 494  
Main() -- : 495  
Main() -- : 496  
Main() -- : 497  
Main() -- : 498  
Main() -- : 499  
Thread : 498  
Thread : 499  
Thread : 500  
D:\java>
```

程序 2

线程的睡眠

需求: 在线程应用 sleep, 让线程执行时进行一段时间的睡眠。

目标:

- 1、sleep 的使用;
- 2、多线程应用。

程序:

```
//: TestMultiThreadSleep.java  
package com.useful.java.part5;  
  
public class TestMultiThreadSleep extends Thread{  
  
    int intNumberThread = 0;  
    int intNumberThreadMain = 0;  
    int intTatol = 500;  
  
    public TestMultiThreadSleep() {  
    }  
  
    public void run(){
```

```
        while(true){
            System.out.println("Thread : " + intNumberThread);
            intNumberThread++;
            //Sleep 0.01 秒钟
            try {
                sleep(10);
            } catch (InterruptedException e){
                e.printStackTrace();
            }
            if(intNumberThread > intTatol){
                return;
            }
        }
    }

    public static void main(String[] args) {
        TestMultiThreadSleep testThread1 = new TestMultiThreadSleep();
        testThread1.start();
        //再加一个线程 testThread2
        TestMultiThreadSleep testThread2 = new TestMultiThreadSleep();
        testThread2.start();
        while(testThread1.intNumberThreadMain < testThread1.intTatol){
            System.out.println("Main()      --      :      "      +
testThread1.intNumberThreadMain);
            testThread1.intNumberThreadMain++;
        }
    }
}
```

说明:

- 1、sleep 只能在线程中使用;
- 2、sleep (1000) 表示睡眠 1 秒, 这里的单位默认为微秒;
- 3、当使用 sleep 时, 需要捕获异常 InterruptedException。

作业

- 1、从 Thread 继承一个类, 并覆盖 run()方法。在 run()内, 打印出一条消息, 然后调用 sleep()。重复三遍这些操作, 然后从 run()返回。在构建器中放置一条启动消息, 并覆盖 finalize(), 打印一条关闭消息。创建一个独立的线程类, 使它在 run()内调用 System.gc()和 System.runFinalization(), 并打印一条消息, 表明调用成功。创建这两种类型的几个线程, 然后运行它们, 看看会发生什么。

第十四章 网络编程

教学目标:

- i 掌握网络编程基本概念
- i 掌握 TCP/IP 编程
- i 掌握 UDP 编程

一：网络编程基本概念

1: 网络通信协议及接口

计算机网络中实现通信必须有一些约定即通信协议，对速率、传输代码、代码结构、传输控制步骤、出错控制等制定标准。而为了使两个结点之间能进行对话，必须在它们之间建立通信工具，即网络通信接口，使彼此之间能进行信息交换。接口包括两部分：

- I 硬件装置：实现结点之间的信息传送
- I 软件装置：规定双方进行通信的约定协议

由于结点之间的联系很复杂，在制定协议时，把复杂成份分解成一些简单的成份，再将它们复合起来。最常用的复合方式是层次方式，即同层间可以通信、上一层可以调用下一层，而与再下一层不发生关系。通信协议的分层原则为，把用户应用程序作为最高层，物理通信线路作为最低层，将其间的协议处理分为若干层，规定每层处理的任务，也规定每层的接口标准，如图所示：

非计算机专业的读者常感困惑的是，在上述各层间进行通信时到底做了什么事情？简单的说，就是进行了数据的封装和拆封：发送方数据在网络模型的各层传送过程中加入头尾说明性信息，而接受方收到数据后去除相应的头尾。就好比张三与李四两人的通信过程：张三将写好的信纸装入信封，在这一封装过程中添加了收信人的邮政编码、地址、姓名等信息；而李四收到信后，要先撕开信封才能读取其中有用的信息。而在此期间，邮局以及运输公司等为保证信件传递的顺利进行，可能还要进行中间层次的封装和拆封工作。

1.1: TCP/IP 协议

TCP (Transmission Control Protocol) /IP (Internet Protocol) 协议是当前网络数据传输的基础协议。他们可保证不同厂家生产的计算机能在共同网络环境下运行，解决异构网通信问题，TCP/IP 与低层的数据链路层和物理层无关，能广泛地支持由低两层协议构成的物理网络结构。TCP -- 面向连接的可靠数据传输协议；TCP 重发一切没有收到的数据，进行数据内容准确性检查并保证数据分组正确顺序。

IP 协议是网际层的主要协议，支持网间互连的数据报通信。它提供主要功能有：无连接数据报传送、数据报路由选择和差错控制。IP 协议主要特性为，IP 协议将报文传送到目的主机后，无论传送正确与否都不进行检验、不回送确认、不保证分组正确顺序。

为实现网络中不同计算机之间的通信，每台机器都必须有一个与众不同的标识，这就是 IP 地址，TCP/IP 使用 IP 地址来标识源地址和目的地址。IP 地址格式：数字型，32 位，由 4 个 8 位的二进制数组成，每 8 位之间用圆点隔开，如：166.111.78.98。

2: URL

URL (统一资源定位器，Uniform Resource Locator) 用于表示 Internet 上资源的地址。这里所说的资源，可以是文件、目录或更为复杂的对象的引用。

URL 一般由协议名、资源所在的主机名和资源名等部分组成，例如下面的 URL：

<http://home.netscape.com/home/welcome.html>

所用的协议是 http (Hypertext Transfer Protocol, 超文本传输协议) 协议, 资源所在的主机名为 home.netscape.com, 资源名为 home/welcome.html。有时资源名也可省略, 这样将指向默认的主页面, 如 <http://www.sun.com>。URL 还可以包含端口号来指定与远端主机相连接的端口。如果不指定端口号, 则使用默认值。例如, http 协议的默认端口号是 80。显式指定端口号的 URL 形式如下:

<http://java.cs.tsinghua.edu.cn:8888/>

java.net 包定义了对应的 URL 类。其常用构造方法及用法举例如下:

```
public URL(String spec);
    URL u1 = new URL("http://home.netscape.com/home/");
public URL(URL context, String spec);
    URL u2 = new URL(u1, "welcome.html");
public URL(String protocol, String host, String file);
    URL u3 = new URL("http", "www.sun.com", "index.html");
public URL (String protocol, String host, int port, String file);
    URL u4 = new URL("http", "www.sun.com", 80, "index.html");
```

使用 URL 类的 openStream() 方法可以建立到当前 URL 的连接并返回一个可用于从该连接读取数据的输入流对象, 方法格式为:

```
public final InputStream openStream() throws IOException
```

示例 使用 URL 读取网络资源程序: URLReader.java

```
import java.io.*;
import java.net.*;
public class URLReader{
    public static void main(String args[]){
        try{
            URL tirc = new URL("http://www.tsinghua.edu.cn/");
            BufferedReader in = new BufferedReader(new
                InputStreamReader(tirc.openStream()));
            String s;
            while((s = in.readLine())!=null)
                System.out.println(s);
            in.close();
        }catch(MalformedURLException e) {
            System.out.println(e);
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

程序运行结果为:

```
<html>
<head>
<title>清华大学网站首页</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<meta http-equiv="refresh" content="0;URL=./eng/index.htm">
</head>
<body bgcolor="#FEFAF2" text="#000000" topmargin=0 leftmargin=0 marginwidth=0
marginheight=0>
</body>
</html>
```

如使用了代理服务器,则应通过添加系统属性的方式指名代理服务器的机器名(或 IP 地址)及端口号。此时运行命令格式如下:

```
java -Dhttp.proxyHost=<hostname|hostIP> -Dhttp.proxyPort=<port> <class_name>
```

例如: java -Dhttp.proxyHost=196.168.100.2 -Dhttp.proxyPort=8129 URLReader

3: 连接的地址

你发起电话呼叫时,你必须知道所拨的电话号码。如果要发起网络连接,你需要知道远程机器的地址或名字。此外,每个网络连接需要一个端口号,你可以把它想象成电话的分机号码。一旦你和一台计算机建立连接,你需要指明连接的目的。所以,就如同你可以使用一个特定的分机号码来和财务部门对话那样,你可以使用一个特定的端口号来和会计程序进行通信。

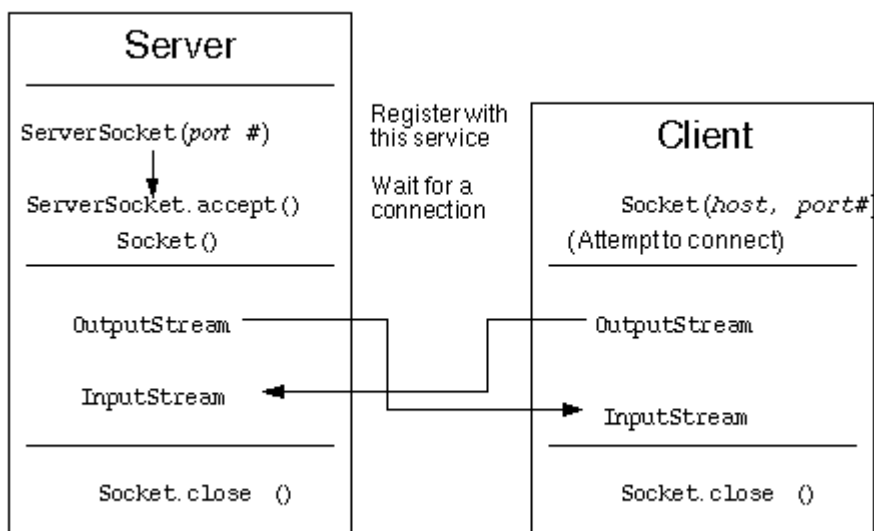
4: 端口号

TCP/IP 系统中的端口号是一个 16 位的数字,它的范围是 0~65535。实际上,小于 1024 的端口号保留给预定义的服务,而且除非要和那些服务之一进行通信(例如 telnet, SMTP 邮件和 ftp 等),否则你不应该使用它们。

客户和服务端必须事先约定所使用的端口。如果系统两部分所使用的端口不一致,那就不能进行通信。

5: Java 网络模型

在 Java 编程语言中, TCP/IP socket 连接是用 java.net 包中的类实现的。下图说明了服务器和客户端所发生的动作。



- I 服务器分配一个端口号。如果客户请求一个连接, 服务器使用 `accept()` 方法打开 socket 连接。
- I 客户在 host 的 port 端口建立连接。
- I 服务器和客户使用 `InputStream` 和 `OutputStream` 进行通信。

二: TCP/IP 编程

1: Socket 基础

计算机以一种非常简单的方式进行相互间的操作和通信。计算机芯片是以 1 和 0 的形式存储并传输数据的开—闭转换器的集合。当计算机想共享数据时, 它们所需做的全部就是以一致的速度、顺序、定时等等来回传输几百万比特和字节的数据流。每次想在两个应用程序之间进行信息通信时, 您怎么会愿意担心那些细节呢?

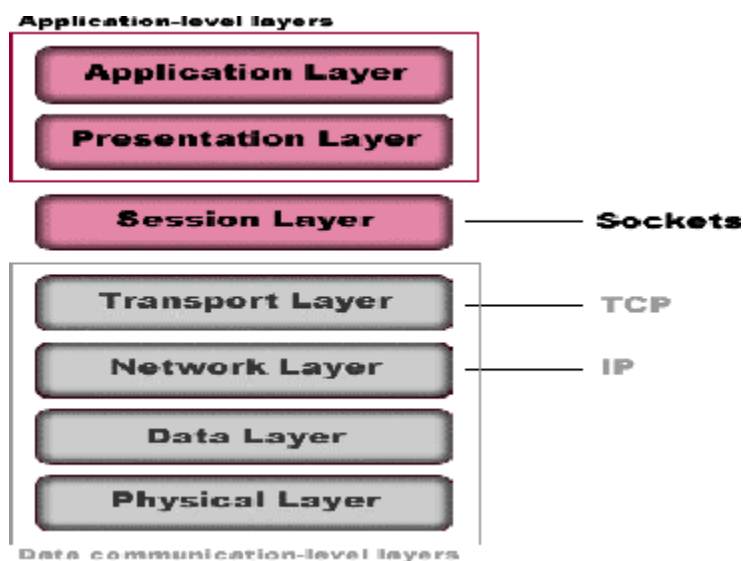
为免除这些担心, 我们需要每次都以相同方式完成该项工作的一组包协议。这将允许我们处理应用程序级的工作, 而不必担心低级网络细节。这些成包协议称为协议栈 (*stack*)。TCP/IP 是当今最常见的协议栈。多数协议栈 (包括 TCP/IP) 都大致对应于国际标准化组织 (International Standards Organization, ISO) 的开放系统互连参考模型 (Open Systems Interconnect Reference Model, OSIRM)。OSIRM 认为在一个可靠的计算机组网中有七个逻辑层 (见图)。各个地方的公司都对这个模型某些层的实现做了一些贡献, 从生成电子信号 (光脉冲、射频等等) 到提供数据给应用程序。

socket 是指在一个特定编程模型下, 进程间通信链路的端点。因为这个特定编程模型的流行, socket 这个名字在其他领域得到了复用, 包括 Java 技术。

当进程通过网络进行通信时, Java 技术使用它的流模型。一个 socket 包括两个流: 一个输入流和一个输出流。如果一个进程要通过网络向另一个进程发送数据, 只需简单地写入与 socket 相关联的输出流。一个进程通过从与 socket 相关联的输入流读来读取另一个进程所写的的数据。

建立网络连接之后, 使用与 socket 相关联的流和使用其他流是非常相似的。

我们不想涉及层的太多细节, 但您应该知道套接字位于什么地方



使用套接字的代码工作于表示层。表示层提供应用层能够使用的信息的公共表示。假设您打算把应用程序连接到只能识别 EBCDIC 的旧的银行系统。应用程序的域对象以 ASCII 格式存储信息。在这种情况下,您得负责在表示层上编写把数据从 EBCDIC 转换成 ASCII 的代码,然后(比方说)给应用层提供域对象。应用层然后就可以用域对象来做它想做的任何事情。

您编写的套接字处理代码只存在于表示层中。您的应用层无须知道套接字如何工作的任何事情。

简言之,一台机器上的套接字与另一台机器上的套接字交谈就创建一条通信通道。程序员可以用该通道来在两台机器之间发送数据。当您发送数据时,TCP/IP 协议栈的每一层都会添加适当的报头信息来包装数据。这些报头帮助协议栈把您的数据送到目的地。好消息是 Java 语言通过“流”为您的代码提供数据,从而隐藏了所有这些细节,这也是为什么它们有时候被叫做**流套接字**(*streaming socket*)的原因。

把套接字想成两端电话上的听筒——我和您通过专用通道在我们的电话听筒上讲话和聆听。直到我们决定挂断电话,对话才会结束(除非我们在使用蜂窝电话)。而且我们各自的电话线路都占线,直到我们挂断电话。

如果想在没有更高级机制如 ORB(以及 CORBA、RMI、IIOP 等等)开销的情况下进行两台计算机之间的通信,那么套接字就适合您。套接字的低级细节相当棘手。幸运的是,Java 平台给了您一些虽然简单但却强大的更高级抽象,使您可以容易地创建和使用套接字。

2: 套接字的类型

一般而言,Java 语言中的套接字有以下两种形式:

- I TCP 套接字(由 Socket 类实现,稍后我们将讨论这个类)
- I UDP 套接字(由 DatagramSocket 类实现)

TCP 和 UDP 扮演相同角色,但做法不同。两者都接收传输协议数据包并将其内容向前传送到表示层。TCP 把消息分解成数据包(数据报, *datagrams*)并在接收端以正确的顺序把它们重新装配起来。TCP 还处理对遗失数据包的重传请求。有了 TCP,位于上层的层要担心的事情就少多了。UDP 不提供装配和重传请求这些功能。它只是向前传送信息包。位于上层的层必须确保消息是完整的并且是以正确的顺序装配的。

一般而言,UDP 强加给您的应用程序的性能开销更小,但只在应用程序不会突然交换大量数据并且不必装配大量数据报以完成一条消息的时候。否则,TCP 才是最简单或许也是最高效的选择。因为多数读者都喜欢 TCP 胜过 UDP,所以我们将把讨论限制在 Java 语言中面向 TCP 的类。

3: 编写简单的 Socket 程序

java.net 包中定义了两个类 Socket 和 ServerSocket,分别用来表示双向连接的客户端和服务端。其常用的构造方法有:

```
Socket(InetAddress address, int port);
```

```
Socket(InetAddress address, int port, boolean stream);
Socket(String host, int port);
Socket(String host, int port, boolean stream);
ServerSocket(int port);
ServerSocket(int port, int count);
```

网络编程的四个基本步骤为:

创建 socket

打开连接到 socket 的输入/输出流

按照一定的协议对 socket 进行读/写操作

关闭 socket

示例 简单的 client/server 程序: TestServer.java

```
import java.net.*;
import java.io.*;

public class TestServer {
    public static void main(String args[]) {
        try {
            ServerSocket s = new ServerSocket(8888);
            while (true) {
                Socket s1 = s.accept();
                OutputStream os = s1.getOutputStream();
                DataOutputStream dos =
new DataOutputStream(os);
                dos.writeUTF("Hello," + s1.getInetAddress() +
"port#" + s1.getPort() + "\nbye!");
                dos.close();
                s1.close();
            }
        } catch (IOException e) {
            System.out.println("程序运行出错:" + e);
        }
    }
}
```

程序: TestClient.java

```
import java.net.*;
import java.io.*;

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

```
try {
    Socket s1 = new Socket("127.0.0.1", 8888);
    InputStream is = s1.getInputStream();
    DataInputStream dis = new DataInputStream(is);
    System.out.println(dis.readUTF());
    dis.close();
    s1.close();
} catch (ConnectException connExc) {
    System.err.println("服务器连接失败! ");
} catch (IOException e) {
}
}
```

TestServer.java 和 TestClient.java 程序可运行在两台不同的机器上, 首先运行 TestServer 程序, 创建 ServerSocket 对象、监听所在机器(服务器)的指定 8888 号端口, 等待客户端连接请求; 然后运行 TestClient 程序, 创建 Socket 对象, 连接服务器的 8888 号端口。建立连接后, 服务器端程序打开其 Socket 对象关联的输出流, 并向其中写出有关连接者的信息, 然后关闭输出流及 Socket 对象、程序退出; 客户端程序打开 Socket 对象关联的输入流, 从中读取服务器端发送来的信息并显示到屏幕上。程序运行结果为:

Hello, zhangliguo/127.0.0.1port#1032 bye-bye!

其中, "zhangliguo/127.0.0.1" 为客户端机器名和 IP 地址, 1032 是客户端由机器指定的发出连接请求的端口号。上述程序也可以运行在同一台计算机上, 此时该机器既是服务器, 又是客户机, 但运行原理是相同的。

4: JAVA 中的 socket 实现

Java 平台在 java.net 包中提供套接字的实现。在本教程中, 我们将与 java.net 中的以下三个类一起工作:

URLConnection 、 Socket 、 ServerSocket

java.net 中还有更多的类, 但这些是您将最经常碰到的。让我们从 URLConnection 开始。这个类为您不必了解任何底层套接字细节就能在 Java 代码中使用套接字提供一种途径 URLConnection。

URLConnection 类是所有在应用程序和 URL 之间创建通信链路的类的抽象超类。URLConnection 在获取 Web 服务器上的文档方面特别有用, 但也可用于连接由 URL 标识的任何资源。该类的实例既可用于从资源中读, 也可用于往资源中写。例如, 您可以连接到一个 servlet 并发送一个格式良好的 XML String 到服务器上进行处理。URLConnection 的具体子类(例如 HttpURLConnection)提供特定于它们实现的额外功能。对于我们的示例, 我们不想做任何特别的事情, 所以我们将使用 URLConnection 本身提供的缺省行为。

连接到 URL 包括几个步骤:

1. 创建 `URLConnection`
2. 用各种 `setter` 方法配置它
3. 连接到 URL
4. 用各种 `getter` 方法与它交互
5. 接着,我们将看一些演示如何用 `URLConnection` 来从服务器请求文档的样本代码。

4.1: URLClient 类

我们将从 `URLClient` 类的结构讲起。

```
import java.io.*;
import java.net.*;
public class URLClient {
    protected URLConnection connection;
    public static void main(String[] args) {
    }
    public String getDocumentAt(String urlString) {
    }
}
```

要做的第一件事是导入 `java.net` 和 `java.io`。我们给我们的类一个实例变量以保存一个 `URLConnection`。我们的类有一个 `main()` 方法,它处理浏览文档的逻辑流。我们的类还有一个 `getDocumentAt()` 方法,该方法连接到服务器并向它请求给定文档。下面我们将分别探究这些方法的细节。

4.2: 浏览文档

`main()` 方法处理浏览文档的逻辑流:

```
public static void main(String[] args) {
    URLClient client = new URLClient();
    String yahoo = client.getDocumentAt("http://www.yahoo.com");
    System.out.println(yahoo);
}
```

我们的 `main()` 方法只是创建一个新的 `URLClient` 并用一个有效的 URL String 调用 `getDocumentAt()`。当调用返回该文档时,我们把它存储在 `String`,然后将它打印到控制台。然而,实际的工作是在 `getDocumentAt()` 方法中完成的。

4.3: 从服务器请求一个文档

`getDocumentAt()` 方法处理获取 Web 上的文档的实际工作:

```
public String getDocumentAt(String urlString) {
    StringBuffer document = new StringBuffer();
    try {
        URL url = new URL(urlString);
        URLConnection conn = url.openConnection();
        BufferedReader reader = new BufferedReader(new
```

```
InputStreamReader(conn.getInputStream()));

    String line = null;
    while ((line = reader.readLine()) != null)
        document.append(line + "\n");
    reader.close();
} catch (MalformedURLException e) {
    System.out.println("Unable to connect to URL: " + urlString);
} catch (IOException e) {
    System.out.println("IOException when connecting to URL: " +
urlString);
}
return document.toString();
}
```

`getDocumentAt()` 方法有一个 `String` 参数, 该参数包含我们想获取的文档的 URL。我们在开始时创建一个 `StringBuffer` 来保存文档的行。然后我们用我们传进去的 `urlString` 创建一个新 URL。接着创建一个 `URLConnection` 并打开它:

```
URLConnection conn = url.openConnection();
```

一旦有了一个 `URLConnection`, 我们就获取它的 `InputStream` 并包装进 `InputStreamReader`, 然后我们又把 `InputStreamReader` 包装进 `BufferedReader` 以使我们能够读取想从服务器上获取的文档的行。在 Java 代码中处理套接字时, 我们将经常使用这种包装技术, 但我们不会总是详细讨论它。在我们继续往前讲之前, 您应该熟悉它:

```
BufferedReader reader=
new BufferedReader(new InputStreamReader(conn.getInputStream()));
```

有了 `BufferedReader`, 就使得我们能够容易地读取文档内容。我们在 `while` 循环中调用 `reader` 上的 `readLine()`:

```
String line = null;
while ((line = reader.readLine()) != null)
    document.append(line+ "\n");
```

对 `readLine()` 的调用将直至碰到一个从 `InputStream` 传入的行终止符(例如换行符)时才阻塞。如果没碰到, 它将继续等待。只有当连接被关闭时, 它才会返回 `null`。在这个案例中, 一旦我们获取一个行 (`line`), 我们就把它连同换行符一起附加 (`append`) 到名为 `document` 的 `StringBuffer` 上。这保留了服务器端上读取的文档的格式。

我们在读完行之后关闭 `BufferedReader`:

```
reader.close();
```

如果提供给 URL 构造器的 `urlString` 是无效的, 那么将抛出 `MalformedURLException`。如果发生了别的错误, 例如当从连接上获取 `InputStream` 时, 那么将抛出 `IOException`。

4.4: 总结

实际上, `URLConnection` 使用套接字从我们指定的 URL 中读取信息(它只是解析成 IP 地址),但我们无须了解它,我们也不关心。但有很多事;我们马上就去看看。

在继续往前讲之前,让我们回顾一下创建和使用 `URLConnection` 的步骤:

用您想连接的资源的有效 URL String 实例化一个 URL (如有问题则抛出 `MalformedURLException`)。

打开该 URL 上的一个连接。

将该连接的 `InputStream` 包装进 `BufferedReader` 以使您能够读取行。

用 `BufferedReader` 读文档。

关闭 `BufferedReader`。

`URLClient` 的代码清单

```
import java.io.*;
import java.net.*;

public class URLClient {
    protected HttpURLConnection connection;
    public String getDocumentAt(String urlString) {
        StringBuffer document = new StringBuffer();
        try {
            URL url = new URL(urlString);
            URLConnection conn = url.openConnection();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String line = null;
            while ((line = reader.readLine()) != null)
                document.append(line + "\n");
            reader.close();
        } catch (MalformedURLException e) {
            System.out.println("Unable to connect to URL: " +
urlString);
        } catch (IOException e) {
            System.out.println("IOException when connecting to URL: "
+ urlString);
        }
        return document.toString();
    }
    public static void main(String[] args) {
        URLClient client = new URLClient();
        String yahoo = client.getDocumentAt("http://www.yahoo.com");
        System.out.println(yahoo);
    }
}
```


三: UDP 编程

UDP socket

- | 它们是无连接的协议。
- | 不保证消息的可靠传输。
- | 它们由 Java 技术中的 DatagramSocket 和 DatagramPacket 类支持。

TCP/IP 是面向连接的协议。而用户数据报协议(UDP)是一种无连接的协议。要区分这两种协议,一种很简单而又很贴切的方法是把它们比作电话呼叫和邮递信件。

电话呼叫保证有一个同步通信;消息按给定次序发送和接收。而对于邮递信件,即使能收到所有的消息,它们的顺序也可能不同。

用户数据报协议(UDP)由 Java 软件的 DatagramSocket 和 DatagramPacket 类支持。包是自包含的消息,它包括有关发送方、消息长度和消息自身。

1: DatagramPacket

DatagramPacket

DatagramPacket 有两个构造函数:一个用来接收数据,另一个用来发送数据。

DatagramPacket(byte [] recvBuf, int readLength)

DatagramPacket(byte [] sendBuf, int sendLength, InetAddress iaddr, int iport)

DatagramPacket 有两个构造函数:一个用来接收数据,另一个用来发送数据:

DatagramPacket(byte [] recvBuf, int readLength)

用来建立一个字节数组以接收 UDP 包。byte 数组在传递给构造函数时是空的,而 int 值用来设定要读取的字节数(不能比数组的大小还大)。

DatagramPacket(byte [] sendBuf, int sendLength, InetAddress iaddr, int iport)

用来建立将要传输的 UDP 包。sendLength 不应该比 sendBuf 字节数组的大小要大。

2: DatagramSocket

DatagramSocket 用来读写 UDP 包。这个类有三个构造函数,允许你指定要绑定的端口号和 internet 地址:

DatagramSocket()—绑定本地主机的所有可用端口

DatagramSocket(int port)—绑定本地主机的指定端口

DatagramSocket(int port, InetAddress iaddr)—绑定指定地址的指定端口

3: 最小 UDP 服务器

最小 UDP 服务器在 8000 端口监听客户的请求。当它从客户接收到一个 DatagramPacket 时,它发送服务器上的当前时间。


```
1.import java.io.*;
2.import java.net.*;
3.import java.util.*;
4.
5.public class UdpServer{
6.
7.//This method retrieves the current time on the server
8.public byte[] getTime(){
9.Date d= new Date();
10.return d.toString().getBytes();
11.}
12.
13.// Main server loop.
14.public void go() throws IOException {
15.
16.DatagramSocket datagramSocket;
17.DatagramPacket inDataPacket; // Datagram packet from the client
18.DatagramPacket outDataPacket; // Datagram packet to the client
19.InetAddress clientAddress; // Client return address
20.int clientPort; // Client return port
21.byte[] msg= new byte[10]; // Incoming data buffer. Ignored.
22.byte[] time; // Stores retrieved time
23.
24.// Allocate a socket to man port 8000 for requests.
25.datagramSocket = new DatagramSocket(8000);
26.System.out.println("UDP server active on port 8000");
27.
28.// Loop forever
29.while(true) {
30.
31.// Set up receiver packet. Data will be ignored.
32.inDataPacket = new DatagramPacket(msg, msg.length);
33.// Get the message.
34.datagramSocket.receive(inDataPacket);
35.
36.// Retrieve return address information, including InetAddress
37.// and port from the datagram packet just recieved.
38.clientAddress = inDataPacket.getAddress();
39.clientPort = inDataPacket.getPort();
40.
41.// Get the current time.
42.time = getTime();
43.
44.//set up a datagram to be sent to the client using the
```

```
45.//current time, the client address and port
46.outDataPacket = new DatagramPacket
47.(time, time.length, clientAddress, clientPort);
48.
49.//finally send the packet
50.datagramSocket.send(outDataPacket);
51.}
52.}
53.
54.public static void main(String args[]) {
55.UdpServer udpServer = new UdpServer();
56.try {
57.udpServer.go();
58.} catch (IOException e) {
59.System.out.println ("IOException occured with socket.");
60.System.out.println (e);
61.System.exit(1);
62.}
63.}
64.}
```

4: 最小 UDP 客户

最小 UDP 客户向前面创建的客户发送一个空包并接收一个包含服务器实际时间的包。

```
1.import java.io.*;
2.import java.net.*;
3.
4.public class UdpClient {
5.
6.public void go() throws IOException, UnknownHostException {
7.DatagramSocket datagramSocket;
8.DatagramPacket outDataPacket; // Datagram packet to the server
9.DatagramPacket inDataPacket; // Datagram packet from the server
10.InetAddress
serverAddress; // Server host
address
11.byte[] msg = new byte[100]; // Buffer space.
12.String receivedMsg; // Received message in String form.
13.
14.// Allocate a socket by which messages are sent and received.
15.datagramSocket = new DatagramSocket();
16.
17.// Server is running on this same machine for this example.
18.// This method can throw an UnknownHostException.
19.serverAddress = InetAddress.getLocalHost();
```

```
20.
21.// Set up a datagram request to be sent to the server.
22.// Send to port 8000.
23.outDataPacket = new DatagramPacket(msg, 1, serverAddress, 8000);
24.
25.// Make the request to the server.
26.datagramSocket.send(outDataPacket);
27.
28.// Set up a datagram packet to receive server's response.
29.inDataPacket = new DatagramPacket(msg, msg.length);
30.
31.// Receive the time data from the server
32.datagramSocket.receive(inDataPacket);
33.
34.// Print the data received from the server
35.receivedMsg = new String
36.(inDataPacket.getData(), 0, inDataPacket.getLength());
37.System.out.println(receivedMsg);
38.
39.//close the socket
40.datagramSocket.close();
41.}
42.
43.public static void main(String args[]) {
44.UdpClient udpClient = new UdpClient();
45.try {
46.udpClient.go();
47.} catch (Exception e) {
48.System.out.println ("Exception occurred with socket.");
49.System.out.println (e);
50.System.exit(1);
51.}
52.}
53.}
```

练习实践

本章的内容为 Socket, 实践重点:

I Socket 基础

程序 1:

用 Socket 通讯

需求: 用 Socket 创建一个服务器及客户端, 并使其通讯。

目标:

- 1、Socket 基础;
- 2、服务器的建立, 客户端的建立;
- 3、通讯方式。

程序 1 (服务器端程序):

```
//: JabberServer.java
import java.io.*;
import java.net.*;

public class JabberServer {
    // Choose a port outside of the range 1-1024:
    public static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Started: " + s);
        try {
            // Blocks until a connection occurs:
            Socket socket = s.accept();
            try {
                System.out.println(
                    "Connection    accepted:  "+
socket);

                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(
                            socket.getInputStream()));

                // Output is automatically flushed
                // by PrintWriter:
                PrintWriter out =
                    new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(
                                socket.getOutputStream()))),true);
```

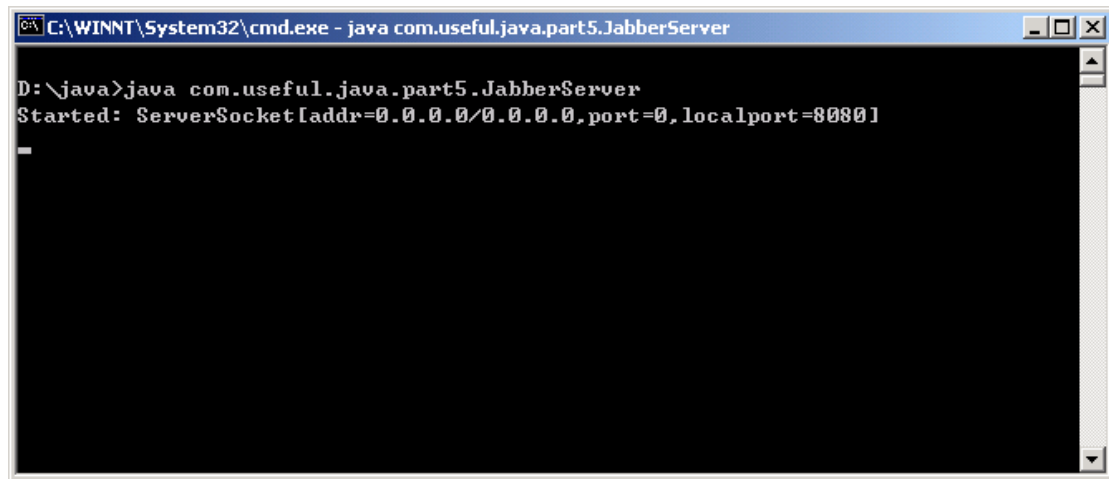
```
        while (true) {
            String str = in.readLine();
            if (str.equals("END")) break;
            System.out.println("Echoing: " + str);
            out.println(str);
        }
        // Always close the two sockets...
    } finally {
        System.out.println("closing...");
        socket.close();
    }
} finally {
    s.close();
}
}
```

说明:

- 1、ServerSocket 需要的只是一个端口编号，不需要 IP 地址（因为它就在这台机器上运行）。调用 accept() 时，方法会暂时陷入停顿状态（堵塞），直到某个客户尝试同它建立连接。换言之，尽管它在那里等候连接，但其他进程仍能正常运行。建好一个连接以后，accept() 就会返回一个 Socket 对象，它是那个连接的代表。
- 2、假如 ServerSocket 构建器失败，则程序简单地退出（注意必须保证 ServerSocket 的构建器在失败之后不会留下任何打开的网络套接字）。针对这种情况，main() 会抛出一个 IOException 违例，所以不必使用一个 try 块。若 ServerSocket 构建器成功执行，则其他所有方法调用都必须到一个 try-finally 代码块里寻求保护，以确保无论块以什么方式留下，ServerSocket 都能正确地关闭。
- 3、同样的道理也适用于由 accept() 返回的 Socket。若 accept() 失败，那么我们必须保证 Socket 不再存在或者含有任何资源，以便不必清除它们。但假若执行成功，则后续的语句必须进入一个 try-finally 块内，以保障在它们失败的情况下，Socket 仍能得到正确的清除。由于套接字使用了重要的非内存资源，所以在这里必须特别谨慎，必须自己动手将它们清除（Java 中没有提供“破坏器”来帮助 我们做这件事情）。
- 4、无论 ServerSocket 还是由 accept() 产生的 Socket 都打印到 System.out 里。这意味着它们的 toString 方法会得到自动调用。这样便产生了：

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

- 5、一般服务器应先启动，启动后，运行如下：



程序 2 (客户端程序):

```
//: JabberClient.java
import java.net.*;
import java.io.*;

public class JabberClient {
    public static void main(String[] args)
        throws IOException {
        // Passing null to getByName() produces the
        // special "Local Loopback" IP address, for
        // testing on one machine w/o a network:
        InetAddress addr =
            InetAddress.getByName(null);
        // Alternatively, you can use
        // the address or name:
        // InetAddress addr =
        //     InetAddress.getByName("127.0.0.1");
        // InetAddress addr =
        //     InetAddress.getByName("localhost");
        System.out.println("addr = " + addr);
        Socket socket =
            new Socket(addr, JabberServer.PORT);
        // Guard everything in a try-finally to make
        // sure that the socket is closed:
        try {
            System.out.println("socket = " + socket);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
```

```
// Output is automatically flushed
// by PrintWriter:
PrintWriter out =
    new PrintWriter(
        new BufferedWriter(
            new OutputStreamWriter(
                socket.getOutputStream()),true);
for(int i = 0; i < 10; i++) {
    out.println("howdy " + i);
    String str = in.readLine();
    System.out.println(str);
}
out.println("END");
} finally {
    System.out.println("closing...");
    socket.close();
}
}
```

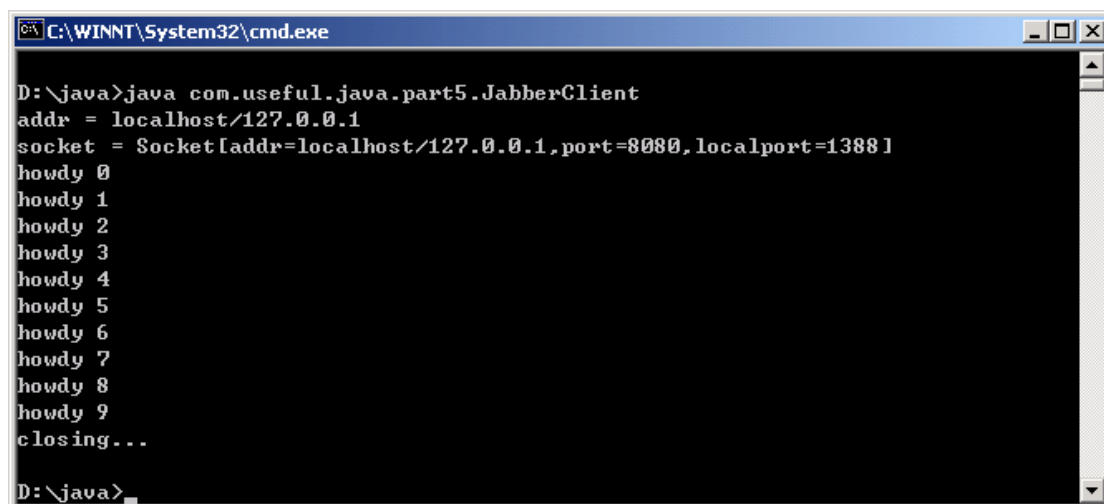
说明:

1. 运行客户端时, 不能关服务器端程序运行窗口, 而应新开一个窗口;
2. 在 main()中, 大家可看到获得本地主机 IP 地址的 InetAddress 的三种途径: 使用 null, 使用 localhost, 或者直接使用保留地址 127.0.0.1。当然, 如果想通过网络同一台远程主机连接, 也可以换用那台机器的 IP 地址。打印出 InetAddress addr 后 (通过对 toString() 方法的自动调用), 结果如下:

localhost/127.0.0.1

通过向 getByName()传递一个 null, 它会默认寻找 localhost, 并生成特殊的保留地址 127.0.0.1。

3. 一次独一无二的因特网连接是用下述四种数据标识的: clientHost (客户主机)、clientPortNumber (客户端口号)、serverHost (服务主机) 以及 serverPortNumber (服务端口号)。服务程序启动后, 会在本地主机 (127.0.0.1) 上建立为它分配的端口 (8080)。一旦客户程序发出请求, 机器上下一个可用的端口就会分配给它 (这种情况下是 1077), 这一行动也在与服务程序相同的机器 (127.0.0.1) 上进行。
4. 现在, 为了使数据能在客户及服务程序之间来回传送, 每一端都需要知道把数据发到哪里。所以在同一个“已知”服务程序连接的时候, 客户会发出一个“返回地址”, 使服务器程序知道将自己的数据发到哪儿。我们在服务器端的示范输出中可以体会到这一情况:
Socket[addr=127.0.0.1,port=1077,localport=8080]
这意味着服务器刚才已接受了来自 127.0.0.1 这台机器的端口 1077 的连接, 同时监听自己的本地端口 (8080)。而在客户端:
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
这意味着客户已用自己的本地端口 1077 与 127.0.0.1 机器上的端口 8080 建立了连接。
5. 客户端程序运行如下:

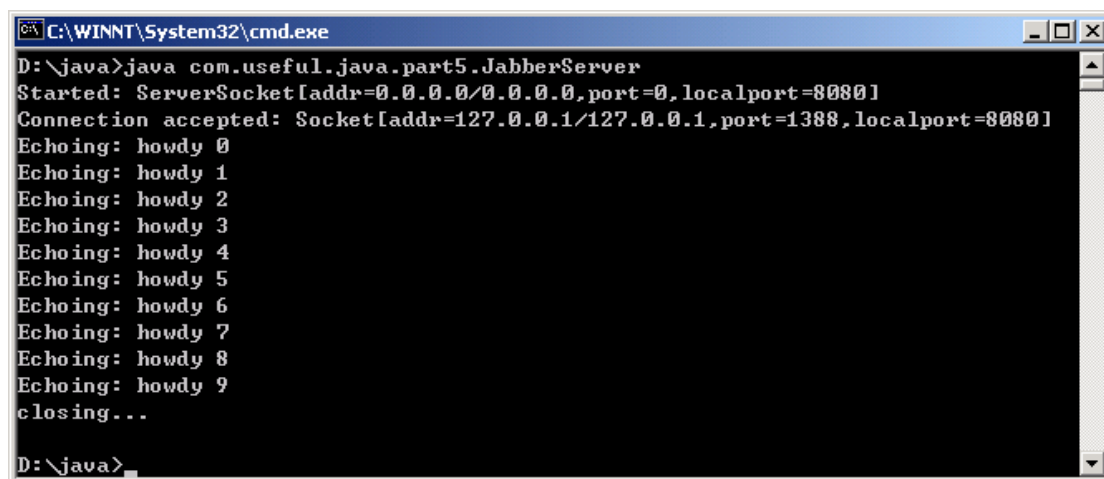


```
C:\WINNT\System32\cmd.exe

D:\java>java com.useful.java.part5.JabberClient
addr = localhost/127.0.0.1
socket = Socket[addr=localhost/127.0.0.1,port=8080,localport=1388]
howdy 0
howdy 1
howdy 2
howdy 3
howdy 4
howdy 5
howdy 6
howdy 7
howdy 8
howdy 9
closing...

D:\java>
```

与此同时，服务器端程序改变如下：



```
C:\WINNT\System32\cmd.exe

D:\java>java com.useful.java.part5.JabberServer
Started: ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=8080]
Connection accepted: Socket[addr=127.0.0.1/127.0.0.1,port=1388,localport=8080]
Echoing: howdy 0
Echoing: howdy 1
Echoing: howdy 2
Echoing: howdy 3
Echoing: howdy 4
Echoing: howdy 5
Echoing: howdy 6
Echoing: howdy 7
Echoing: howdy 8
Echoing: howdy 9
closing...

D:\java>
```

作业

- 1: 修改程序 1，删去为输入和输出设计的所有缓冲机制，然后再次编译和运行，观察一下结果。
- 2: 修改程序 2JabberClient，禁止输出刷新，并观察结果。
- 3: 可以尝试做一个类似 QQ 的聊天通讯工具

第十五章 反射和注解

教学目标:

- i 掌握反射基本概念□
- i 掌握反射编程□
- i 理解注解概念
- i 掌握常见注解的使用
- i 熟悉自定义注解

一: 反射

1: 什么是反射

反射(Reflection)是 Java 程序开发语言的特征之一,它允许运行中的 Java 程序对自身进行检查,或者说“自审”,并能直接操作程序的内部属性。例如,使用它能获得 Java 类中各成员的名称并显示出来。

Java 的这一能力在实际应用中应用得很多,在其它的程序设计语言中根本就不存在这一特性。例如,Pascal、C 或者 C++ 中就没有办法在程序中获得函数定义相关的信息。

比如后面会学到的 JavaBean 是 reflection 的实际应用之一,它能让一些工具可视化的操作软件组件。这些工具通过 reflection 动态的载入并取得 Java 组件(类)的属性。还有后面学习的各种框架,基本上都会有反射的使用。

2: 反射的第一个例子

考虑下面这个简单的例子,让我们看看 reflection 是如何工作的。

```
import java.lang.reflect.*;
public class Test {
    public static void main(String args[]) {
        try {
            Class c = Class.forName("java.util.Stack");
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++)
                System.out.println(m[i].toString());
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

它的结果输出为:

```
public synchronized java.lang.Object java.util.Stack.pop()
public java.lang.Object java.util.Stack.push(java.lang.Object)
public boolean java.util.Stack.empty()
public synchronized java.lang.Object java.util.Stack.peek()
public synchronized int java.util.Stack.search(java.lang.Object)
```

这样就列出了 java.util.Stack 类的各方法名以及它们的限制符和返回类型。

这个程序使用 Class.forName 载入指定的类,然后调用 getDeclaredMethods 来获取这个类中定义了的方法列表。java.lang.reflect.Methods 是用来描述某个类中单个方法的一个类。

3: 开始使用反射

用于反射的类,如 Method,可以在 java.lang.reflect 包中找到。使用这些类的时候必须要遵循三个步骤:

第一步: 是获得你想操作的类的 java.lang.Class 对象。在运行中的 Java 程序中,用 java.lang.Class 类来描述类和接口等。

下面就是获得一个 Class 对象的方法之一:

```
Class c = Class.forName("java.lang.String");
```

这条语句得到一个 String 类的类对象。还有另一种方法, 如下面的语句:

```
Class c = int.class;
```

或者

```
Class c = Integer.TYPE;
```

它们可获得基本类型的类信息。其中后一种方法中访问的是基本类型的封装类 (如 Integer) 中预先定义好的 TYPE 字段。

第二步: 调用诸如 `getDeclaredMethods` 的方法, 取得该类中定义的所有方法的列表

第三步: 使用 反射的 API 来操作这些信息, 如下面这段代码:

```
Class c = Class.forName("java.lang.String");
```

```
Method m[] = c.getDeclaredMethods();
```

```
System.out.println(m[0].toString());
```

它将以文本方式打印出 String 中定义的第一个方法的原型。

在下面的例子中, 这三个步骤将为使用 reflection 处理特殊应用程序提供例证。

示例: 模拟 instanceof 操作符

得到类信息之后, 通常下一个步骤就是解决关于 Class 对象的一些基本的问题。例如, `Class.isInstance` 方法可以用于模拟 instanceof 操作符:

```
package cn;
import java.lang.reflect.*;
public class Test {
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.A");
            boolean b1 = cls.isInstance(new Integer(37));
            System.out.println(b1);
            boolean b2 = cls.isInstance(new A());
            System.out.println(b2);
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
class A {
}
```

在这个例子中创建了一个 A 类的 Class 对象, 然后检查一些对象是否是 A 的实例。Integer(37) 不是, 但 new A() 是。

4: 获取类的方法

找出一个类中定义了些什么方法, 这是一个非常有价值也非常基础的反射用法。下面的代码就实现了这一用法:

```
package cn;
```

```
import java.lang.reflect.*;
```

```
public class Test {
    private int f1(Object p, int x) throws NullPointerException {
        if (p == null)
            throw new NullPointerException();
        return x;
    }
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.Test");
            Method methlist[] = cls.getDeclaredMethods();
            for (int i = 0; i < methlist.length; i++) {
                Method m = methlist[i];
                System.out.println("name = " + m.getName());
                System.out.println("decl class = " + m.getDeclaringClass());
                Class pvec[] = m.getParameterTypes();
                for (int j = 0; j < pvec.length; j++)
                    System.out.println("param #" + j + " " + pvec[j]);
                Class evec[] = m.getExceptionTypes();
                for (int j = 0; j < evec.length; j++)
                    System.out.println("exc #" + j + " " + evec[j]);
                System.out.println("return type = " + m.getReturnType());
                System.out.println("-----");
            }
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

这个程序首先取得 Test 类的描述, 然后调用 `getDeclaredMethods` 来获取一系列的 Method 对象, 它们分别描述了定义在类中的每一个方法, 包括 public 方法、protected 方法、package 方法和 private 方法等。如果你在程序中使用 `getMethods` 来代替 `getDeclaredMethods`, 你还能获得继承来的各个方法的信息。

取得了 Method 对象列表之后, 要显示这些方法的参数类型、异常类型和返回值类型等就不难了。这些类型是基本类型还是类类型, 都可以由描述类的对象按顺序给出。

输出的结果如下:

```
name = f1
decl class = class cn.Test
param #0 class java.lang.Object
param #1 int
exc #0 class java.lang.NullPointerException
return type = int
-----
name = main
decl class = class cn.Test
```

```

param #0 class [Ljava.lang.String;
return type = void
-----

```

5: 获取构造器

获取类构造器的用法与上述获取方法的用法类似, 如:

```

package cn;
import java.lang.reflect.*;
public class Test {
    public Test() {}
    protected Test(int i, double d) {
    }
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.Test");
            Constructor ctorlist[] = cls.getDeclaredConstructors();
            for (int i = 0; i < ctorlist.length; i++) {
                Constructor ct = ctorlist[i];
                System.out.println("name = " + ct.getName());
                System.out.println("decl class = " +
ct.getDeclaredClass());
                Class pvec[] = ct.getParameterTypes();
                for (int j = 0; j < pvec.length; j++)
                    System.out.println("param #" + j + " " + pvec[j]);
                Class evec[] = ct.getExceptionTypes();
                for (int j = 0; j < evec.length; j++)
                    System.out.println("exc #" + j + " " + evec[j]);
                System.out.println("-----");
            }
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}

```

这个例子中没能获得返回类型的相关信息, 那是因为构造器没有返回类型。

这个程序运行的结果是:

```

name = cn.Test
decl class = class cn.Test
-----
name = cn.Test
decl class = class cn.Test
param #0 int
param #1 double
-----

```

6: 获取类的属性字段（域字段）

找出一个类中定义了哪些数据字段也是可以的，如下示例代码：

```
package cn;
import java.lang.reflect.*;
public class Test {
    private double d;
    public static final int i = 37;
    String s = "testing";
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.Test");
            Field fieldlist[] = cls.getDeclaredFields();
            for (int i = 0; i < fieldlist.length; i++) {
                Field fld = fieldlist[i];
                System.out.println("name = " + fld.getName());
                System.out.println("decl class = " + fld.getDeclaringClass());
                System.out.println("type = " + fld.getType());
                int mod = fld.getModifiers();
                System.out.println("modifiers = " + Modifier.toString(mod));
                System.out.println("-----");
            }
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

这个例子和前面那个例子非常相似。例中使用了一个新东西 `Modifier`，它也是一个反射类，用来描述字段成员的修饰语，如“`private int`”。这些修饰语自身由整数描述，而且使用 `Modifier.toString` 来返回以“官方”顺序排列的字符串描述（如“`static`”在“`final`”之前）。这个程序的输出是：

```
name = d
decl class = class cn.Test
type = double
modifiers = private
-----
name = i
decl class = class cn.Test
type = int
modifiers = public static final
-----
name = s
decl class = class cn.Test
type = class java.lang.String
modifiers =
```

和获取方法的情况一下, 获取字段的时候也可以只取得在当前类中声明了的字段信息 (getDeclaredFields), 或者也可以取得父类中定义的字段 (getFields)。

7: 创建新对象实例

对于构造器, 则不能像执行方法那样进行, 因为执行一个构造器就意味着创建了一个新的对象 (准确的说, 创建一个对象的过程包括分配内存和构造对象)。所以, 与上例最相似的例子如下:

```
package cn;
import java.lang.reflect.*;
public class Test {
    public Test() {
    }
    public Test(int a, int b) {
        System.out.println("a = " + a + " b = " + b);
    }
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.Test");
            Class partypes[] = new Class[2];
            partypes[0] = Integer.TYPE;
            partypes[1] = Integer.TYPE;
            Constructor ct = cls.getConstructor(partypes);
            Object arglist[] = new Object[2];
            arglist[0] = new Integer(37);
            arglist[1] = new Integer(47);
            Object retobj = ct.newInstance(arglist);
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

根据指定的参数类型找到相应的构造函数并执行它, 以创建一个新的对象实例。使用这种方法可以在程序运行时动态地创建对象, 而不是在编译的时候创建对象, 这一点非常有价值。

8: 根据方法名称执行方法

前面所举的例子无一例外都与如何获取类的信息有关。我们也可以用反射来做一些其它的事情, 比如执行一个指定了名称的方法。下面的示例演示了这一操作:

```
package cn;
import java.lang.reflect.*;
public class Test {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```

    }
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.Test");
            Class partypes[] = new Class[2];
            partypes[0] = Integer.TYPE;
            partypes[1] = Integer.TYPE;
            Method meth = cls.getMethod("add", partypes);
            //先创建对象实例
            Object obj = cls.newInstance();
            //然后调用方法
            Object arglist[] = new Object[2];
            arglist[0] = new Integer(37);
            arglist[1] = new Integer(47);
            Object retobj = meth.invoke(obj, arglist);
            Integer retval = (Integer) retobj;
            System.out.println(retval.intValue());
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}

```

假如一个程序在执行的某处的时候才知道需要执行某个方法, 这个方法的名称是在程序的运行过程中指定的 (例如, JavaBean 开发环境中就会做这样的事), 那么上面的程序演示了如何做到。

上例中, `getMethod` 用于查找一个具有两个整型参数且名为 `add` 的方法。找到该方法并创建了相应的 `Method` 对象之后, 在正确的对象实例中执行它。执行该方法的时候, 需要提供一个参数列表, 这在上例中是分别包装了整数 37 和 47 的两个 `Integer` 对象。执行方法的返回的同样是一个 `Integer` 对象, 它封装了返回值 84。

9: 修改属性字段的值

反射还有一个用处就是改变对象数据字段的值, 反射可以从正在运行的程序中根据名称找到对象的字段并改变它, 下面的例子可以说明这一点:

```

package cn;
import java.lang.reflect.*;
public class Test {
    public double d;
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("cn.Test");
            Field fld = cls.getField("d");
            Object obj = cls.newInstance();
            System.out.println("d = " + ((Test) obj).d);
            fld.setDouble(obj, 12.34);
        }
    }
}

```



```
        System.out.println("d = " + ((Test) obj).d);
    } catch (Throwable e) {
        System.err.println(e);
    }
}
}
```

这个例子中, 字段 d 的值被变为了 12.34。

10: 操作数组

数组在 Java 语言中是一种特殊的类类型, 一个数组的引用可以赋给 Object 引用。观察下面的例子看看数组是怎么工作的:

```
public class Test {
    public static void main(String args[]) {
        try {
            Class cls = Class.forName("java.lang.String");
            Object arr = Array.newInstance(cls, 10);
            Array.set(arr, 5, "this is a test");
            String s = (String) Array.get(arr, 5);
            System.out.println(s);
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

例中创建了 10 个单位长度的 String 数组, 为第 5 个位置的字符串赋了值, 最后将这个字符串从数组中取得并打印了出来。

下面这段代码提供了一个更复杂的例子:

```
public class Test {
    public static void main(String args[]) {
        int dims[] = new int[] { 5, 10, 15 };
        Object arr = Array.newInstance(Integer.TYPE, dims);
        Object arrobj = Array.get(arr, 3);
        Class cls = arrobj.getClass().getComponentType();
        System.out.println(cls);
        arrobj = Array.get(arrobj, 5);
        Array.setInt(arrobj, 10, 37);
        int arrcast[][][] = (int[][][]) arr;
        System.out.println(arrcast[3][5][10]);
    }
}
```

例中创建了一个 5 x 10 x 15 的整型数组, 并为处于 [3][5][10] 的元素赋了值为 37。注意, 多维数组实际上就是数组的数组, 例如, 第一个 Array.get 之后, arrobj 是一个 10 x 15 的数组。进而取得其中的一个元素, 即长度为 15 的数组, 并使用 Array.setInt 为它的第 10 个元素赋值。

注意创建数组时的类型是动态的, 在编译时并不知道其类型。

二: 注解

1: 元数据

所谓元数据就是数据的数据。也就是说,元数据是描述数据的。就象数据表中的字段一样,每个字段描述了这个字段下的数据的含义。

元数据可以用于创建文档,跟踪代码中的依赖性,甚至执行基本编译时检查。许多元数据工具,如 XDoclet (后面会学到),将这些功能添加到核心 Java 语言中,暂时成为 Java 编程功能的一部分。

一般来说,元数据的好处分为三类:文档编制、编译器检查和代码分析。代码级文档最常被引用。元数据提供了一种有用的方法来指明方法是否取决于其他方法,它们是否完整,特定类是否必须引用其他类,等等。

2: 什么是注解

Java 中的注解就是 Java 源代码的元数据,也就是说注解是用来描述 Java 源代码的。

基本语法就是@后面跟注解的名称。Java5 以上的版本都支持,有预定义的注解,也可以自定义注解。

3: Java 中预定义注解

这里只讲述 java.lang 包当中的三个预定注解

3.1: Override

这个注解的作用是标识某一个方法是否正确覆盖了它的父类的方法。可以用来避免写代码的时候疏忽大意,导致无谓的错误。示例如下:

```
public class UserModel{
    private String userId,name;
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((userId == null) ? 0 : userId.hashCode());
        return result;
    }
}
```

```

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    final UserModel other = (UserModel) obj;
    if (userId == null) {
        if (other.userId != null)
            return false;
    } else if (!userId.equals(other.userId))
        return false;
    return true;
}
}

```

3.2: Deprecated

这个注解的作用是：表示已经不建议使用这个类成员了。

这个注解是一个标记注解。所谓标记注解，就是在源程序中加入这个标记后，并不影响程序的编译，但有时编译器会显示一些警告信息。

如果某个类成员的提示中出现了个词，就表示已经不建议使用这个类成员了。因为这个类成员在未来的 JDK 版本中可能被删除。之所以在现在还保留，是因为给那些已经使用了这些类成员的程序一个缓冲期。如果现在就去掉了，那么这些程序就无法在新的编译器中编译了。

示例如下：

```

public class UserModel{
    private String userId,name;
    @Deprecated
    public String getUserId() {
        return userId;
    }
    .....
}

```

3.3: SuppressWarnings

这个注解的作用是：用来抑制警告信息。

例如，在 Java5 以上编译下面这段代码：

```

import java.util.*;
public class UserModel{

```

```
public static void main(String[] args) {  
    Collection col = new ArrayList();  
}  
}
```

你会得到如下的警告(Warnings):

ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

如果把代码修改成如下, 再次编译, 那么就不会有警告信息了

```
import java.util.*;  
public class UserModel{  
    @SuppressWarnings(value={"unchecked"})  
    public static void main(String[] args) {  
        Collection col = new ArrayList();  
    }  
}
```

4: 自定义注解

自定义注解的语法很简单, 跟定义接口类似, 只是在名称前面加上@符号。

4.1: 最简单的自定义注解

```
public @interface MyAnno {  
}
```

使用这个注解的代码如下:

```
@MyAnno  
public class UserModel{  
}
```

4.2: 添加成员

注解是用来描述源代码的数据, 所以通常需要为注解提供数据成员。定义数据成员后不需要分别定义访问和修改的方法。相反, 只需要定义一个方法, 以成员的名称命名它。数据类型应该是该方法返回值的类型。

示例如下:

```
public @interface MyAnno {  
    public String schoolName();  
}
```

那么这个时候使用的代码如下:

```
@MyAnno(schoolName="Java私塾")  
public class UserModel{  
}
```

4.3: 设置默认值

有时候并不需要用户一定要提供注解的值, 可以为注解设置默认值。如下:

```
public @interface MyAnno {  
    public String schoolName() default "Java私塾";  
}
```

那么这个时候使用的代码如下:

```
@MyAnno
```

```
public class UserModel{
}
```

当然也可以设置 schoolName 的值, 如下也可

```
@MyAnno(schoolName="Java私塾222")
```

```
public class UserModel{
}
```

5: 对注解的注解

自定义注解都是有目的的, 那么如何表示这个目的呢? 那就是对注解进行描述, 也就是注解的注解。

5.1: 指定目标 Target

最明显的元注释就是允许何种程序元素具有定义的注解类型。毫不奇怪, 这种元注释被称为 Target。但是在了解如何使用 Target 之前, 您还需要认识另一个类, 该类被称为 ElementType, 它实际上是一个枚举。这个枚举定义了注释类型可应用的不同程序元素。

```
package java.lang.annotation;
```

```
public enum ElementType {
    TYPE,                // Class, interface, or enum (but not annotation)
    FIELD,                // Field (including enumerated values)
    METHOD,                // Method (does not include constructors)
    PARAMETER,            // Method parameter
    CONSTRUCTOR,          // Constructor
    LOCAL_VARIABLE,       // Local variable or catch clause
    ANNOTATION_TYPE,      // Annotation Types (meta-annotations)
    PACKAGE               // Java package
}
```

ElementType 的枚举值意义很明确, 使用 Target 元注释时, 至少要提供这些枚举值中的一个并指出注释的注释可以应用的程序元素。

示例如下:

```
import java.lang.annotation.*;
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnno {
    public String schoolName() default "Java私塾";
}
```

这说明我们自定义的注解可以用在类上、接口上、枚举上和方法上, 如果没有定义 ElementType.TYPE, 那么 UserModel 是会编译出错的。

通过制定 Target, 就能防止别人误用自定义的注解。

5.2: 设置保持性 Retention

这个元注释和 Java 编译器处理注解的注解类型的方式有关。编译器有几种不同选择:

- 将注解保留在编译后的类文件中, 并在第一次加载类时读取它。
- 将注解保留在编译后的类文件中, 但是在运行时忽略它。

- 按照规定使用注解,但是并不将它保留到编译后的类文件中。

这三种选项用 `java.lang.annotation.RetentionPolicy` 枚举表示,如下所示:

```
package java.lang.annotation;
```

```
public enum RetentionPolicy {  
    SOURCE, // Annotation is discarded by the compiler  
    CLASS, // Annotation is stored in the class file, but ignored by the VM  
    RUNTIME // Annotation is stored in the class file and read by the VM  
}
```

继续修改我们的自定义注解,使用 `Retention` 如下:

```
import java.lang.annotation.*;  
@Retention(RetentionPolicy.SOURCE)  
@Target({ElementType.TYPE, ElementType.METHOD})  
public @interface MyAnno {  
    public String schoolName() default "Java私塾";  
}
```

如果要将保持性设为 `RetentionPolicy.CLASS`,那么什么也不需要做,因为这就是默认行为。

注意: 要想使用反射得到注释信息,须用 `@Retention(RetentionPolicy.RUNTIME)` 进行注解

5.3: 添加公共文档 Documented

在默认的情况下在使用 `javadoc` 自动生成文档时,注解将被忽略掉。如果想在文档中也包含注解,必须使用 `Documented` 为文档注解。

示例如下:

```
import java.lang.annotation.*;  
@Documented  
@Retention(RetentionPolicy.SOURCE)  
@Target({ElementType.TYPE, ElementType.METHOD})  
public @interface MyAnno {  
    public String schoolName() default "Java私塾";  
}
```

5.4: 设置继承 Inherited

在默认的情况下,父类的注解并不会被子类继承。如果要继承,就必须加上 `Inherited` 注解。

示例如下:

```
import java.lang.annotation.*;  
@Inherited  
@Documented  
@Retention(RetentionPolicy.SOURCE)  
@Target({ElementType.TYPE, ElementType.METHOD})  
public @interface MyAnno {  
    public String schoolName() default "Java私塾";  
}
```

6: 如何读取注解

要读取注解的内容, 就需要使用反射的技术。这里简单演示一下:

注意: 要想使用反射得到注释信息, 须用 `@Retention(RetentionPolicy.RUNTIME)` 进行注解, 所以要把上面的 `MyAnno` 的注解修改一下。

```
import java.lang.reflect.*;

public class TestMyAnno {
    public static void main(String[] args) throws Exception {
        Class c = Class.forName("cn.UserModel");
        boolean flag = c.isAnnotationPresent(MyAnno.class);
        System.out.println(flag);
        if(flag){
            MyAnno ma = (MyAnno)c.getAnnotation(MyAnno.class);
            System.out.println("学校名称: =="+ma.schoolName());
            //获取到了这些数据过后, 下面就可以开始你的处理了
        }
    }
}
```

作业

1: 准备工作:

定义一个 `Model` 类, 里面所有的属性都是 `private` 的, 然后为每个属性提供 `getter` 和 `setter` 方法;

再准备一个 `Map`, `map` 的 `key` 值都是类里面的属性字段的字符串表示, 值任意。

真正的工作:

设计一个方法 `Object getModel(Map map, Class cls)`, 传入一个包含所有值的 `Map`, 然后再传入 `Model` 类的 `class`, 那么返回 `Model` 类的实例, 这个实例里面已经包含好了所有相关的数据。也就是把 `Map` 中的数据通过反射, 设置回到 `Model` 类实例中。