JAVA 语言程序设计(连载三)

孟焱　杨乔林　　(中科院计算所)

(续第二篇 Java 语法及其特点)

Java 程序中的变量声明、表达式和控制流语句基本上沿用了 C 的语法规则,不再一一介绍,需要特别提出的是 Java 的异常处理(exception hangdling)和有关语法。

我们已经注意到在 Java 方法声明中出现了一个 throws 关键字,throws 分句是方法的公共接口的一部分,它后面列出了在当前方法中可能发生的"异常"(exception)的类型。对这些异常情况做出适当的处理是方法调用者(包括直接调用者和间接调用者)的责任。

Java 很重视异常处理,在其 API 库中有一个对应于异常情况的类 Throwable,它有许多代表各种具体异常的子类,这些子类常常以"exception"作为名字的后缀,如:ArrayIndexOutOfBoundException 代表数组下标越界。当程序运行中出现异常情况时,Java 运行系统会产生一个类型与之相对应的 Throwable 对象,该对象沿 Java 运行栈逐级往回传递,直到找到一个能够处理这种异常情况的"异常处理程序段"(exception handler)为止。在异常得到处理后,程序回到发生异常处继续执行。用户程序也可利用 throw(注意区别"throws"关键字)语句来人为造成异常,如"throw new IllegalAccessException();"使 Java 运行系统产生对应于"非法访问"的异常处理动作。

Java 中最常使用的异常处理方法是"try - catch - finally"结构,其语法描述为:

"try"语句

("catch""(异常类型 标识符")"语句)*

('finally'语句)?

其中的"语句"可以是由"{}"括起的语句块。关键字"try"后紧跟运行时可能发生异常的语句段,该段就是 catch 分句的作用范围。catch 分句可以有多条,每条给出它所处理的异常的类型以及相应的处理动作,例如打印出错提示信息等。当 try 后的语句在运行时出现某种异常时,Java 运行系统将试图找到一个与之匹配的 catch 分句来处理它。若无法找到匹配的 catch 分句,而同时又缺省 finally 分句,则该异常被传递给外层方法(即本方法的调用者)。这一过程不断重复直到获得一个匹配的 catch 分句或者遇到 finally 分句为止,否则当前程序将终止运行。若 finally 分句不缺省,则只要 try 语句块中出现异常,无论是否找到相应的 catch 分句,finally 后面的语句都将被执行。因此该分句适于用来对付"漏网之鱼"或者做些通用的恢复工作。

第三篇 Java 应用编程接口

一、API 概述

Java 应用编程接口(API)是 Java 开发/运行环境中不可缺少的组成部分,从用户的角度看,API 由一组程序包构成,在这些包中含有 Java 开发和运行环境所需的基本类声明(在本文中,我们把 interface 和 class 统称为"类")。其中有些是 Java 的核心类,它们所代表的语言概念是 Java 编程的基础。此外,API 还为程序员提供了用于实现输入输出、系统属性访问、网络资源访问、安全检查、用户界面等功能的机制,并定义了大量的常用数据类型和数据结构。

Java API 有时也被称为 Java 的"类库",但我们应当意识到它和 C++类库的区别:首先,Java 对 API 的依赖性要比 C++ 对其类库的依赖性大得多。我们不必使用类库就可以成功地编译并运行 C++ 应用程序,而一个 Java 程序(包括 applet)从编译到解释执行都离不开 API 的支持;其次,当类库中的类被引用时,相应的代码在编译时就被链接到可执行文件中,而 Java API 中的类是在运行时载入的。

不同版本的 Java 开发环境,其 API 程序包的数量和内容也不完全相同,但某些基本程序包是必不可少的,如 java.lang, java.applet, java.io, java.util 等,其中最重要的一个是 java.lang,这里我们就重点介绍它。

二、java.lang 程序包

这个程序包与 Java 语言本身密切相关("lang"就是"language"的字头)。Java 的许多核心概念如 Object, Thread, Exception 等都可以在这个包里的找到相应的类声明。以下是其中较为重要的几个:

1. Object

我们已经知道 Object 是 Java 继承树的根，是除简单类型外所有 Java 类型的最终祖先。因而其中的方法也是所有 Java 类共有的。现简介如下：

. clone()：产生当前对象的一个"复制品"。

. copy(Object)：拷贝另一个对象的内容(即实例变量的值)。

. equals(Object)：判断两对象是否相等。

. hashcode()：返回当前对象的"哈希码"，这是一个 32bit 整数，不同对象的"哈希码"一般不同，故可用来构造哈希函数。由于 Java 允许任何类型的对象作为哈希表的关键字，引入这一方法可方便对哈希表的操作。

. getClass()：这是一个非常有用的方法，它返回当前对象所属类型的描述符(descriptor)。有关类描述符的详细介绍参见后面的"class"条目。

. toString() 返回当前对象的字符串表示。

此外，Object 还提供了一组用于实现线程间同步的方法，主要有：notify()，notifyAll()，wait()等。我们将在后面的 Thread 条目中详细讨论它们。

2. Class

以大写字母开头的"Class"与 Java 的关键字"class"完全不同，这里我们所说的是 API 中的一个类的名字。Class 类型的对象包含一个类的运行时表示，因此也被称为"类描述符"(descriptor)。Java 运行系统为它所识别的每一个类建立一个描述符，其中包含了有关这个类型的全部重要信息和操作。一个类描述符就是一个 Class 型的对象。这里要强调：类描述符"描述"的是某个类型的整体，而不是其个别实例。前面提到过的 getClass() 方法就可以用来获得一个对象所属类的描述符。这一方法使我们能够获取任何 Java 对象的类型信息。

下面列出 Class 中的几个重要方法：

. forName(Sting)：静态方法，返回由 String 指定的类的描述符。这个方法的返回类型也是 Class。

. getClassLoader()：返回该类的"类装载器"(class loader)。类装载器指定一个类以何种方式载入运行系统。缺省方式是从本地文件系统的 Java 工作路径中读入 .class 文件。

. getInterfaces()：返回类中所包含的接口。此方法返回类型为 Class[]，是一个由接口的描述符组成的数组，这主要是考虑到一个类可能实现多个接口。

. getName()：返回该类的名字字符串。

. getSuperclass()：返回基类的描述符。

. newInstance()：创建该类的一个新的实例，返回类型为 Object。

3. String 和 StringBuffer

Java 开发环境提供两个类来实现字符串数据的存储和修改，它们是 String 和 StringBuffer。其中 String 对应字符串常量，StringBuffer 对应可变字符串。Java 中的字符串是第一类对象，这与 C 语言中把字符串简单的作为字符数组来处理有着很大的不同。

把字符串分成常量型和可变型，有利于提高程序的可靠性和整体性能，也符合人们的编程习惯。程序员常常要求某些重要的字符串数据能保持其值，不会被意外地修改或丢失。当这些字符串是共享数据时，这一要求就显得尤为迫切。常见的一种错误是：调用者把字符串作为参数传递给某段代码，然而等到返回时，此字符串已被改得面目全非。对于依靠字符数组的首指针来传递字符串参数的 C 语言来说，这种错误是很难避免的，而在 Java 中只需要使用 String 类型即可，一个 String 型的字符串不可以被修改，因而可以放心地把它作为参数来传递。它也可以被多个并发执行的线程所共享，而不会引起任何副作用。当我们要求字符串的内容可变时，就应当使用 StringBuffer，对这个类型的字符串可以任意修改其内容和长度。如果必要的话，甚至可以让这种字符串不停地"生长"直到耗尽系统内存。

String 和 StringBuffer 都是依靠一个名为 value 的私有字符数组来存放字符串数据的。不同的是后者的 value 可以改变，而前者不能。StringBuffer 可以通过方法 toString ()转换为 String 类型，此时 Java 并不产生 value 数组的一个拷贝，而只是将它标记为"共享的"，即所产生的 String 对象和原有的 tringBuffer 对象共用同一个 value 数组。当程序试图再修改原来的 StringBuffer 对象时，由于其 value 已是共享的，运行系统将生成该数组的一个完全相同的拷贝，并将其作为 StringBuffer 对象的新 value 数组，以后的修改操作全部在新数组上进行，原有字符串数据因此不会再被改变。我们仍然可以通过 已有的 String 型的对象来访问它。

由于 String 的实现机制较 StringBuffer 简单，因此其操作代价较小，建议在无需改变字符串内容的场合，

尽量用 String 类型。

Java 编译器为它遇到的每个串字面量创建一个 String 对象。例如下面的程序行是合法的：

int len = "Hello world!".length();

这里的 "Hello world!" 就是一个 String 对象，lengh() 是 String 的一个常用方法，其作用是返回字符串的长度。我们还可以在字符串表达式中使用"+"操作符以实现串的连接，如：

"This is a " + " string"

此时编译器将其替换为下面的代码：

new StringBuffer. append(" This is a"). append(" string"). toString()其中 append()方法的功能是将括号内的字符串作为后缀加到原来的 StringBuffer 对象上。

4. Thread

Java 线程是由 Thread 这个类来实现的。Thread 规定了 Java 线程所具有的系统无关的公共属性和行为。线程有时也称为"轻量级进程"(lightweight process)或"执行上下文"(execection context)，这里将它定义为存在于一个程序中的单个顺序控制流。一个 Java 程序在运行时可以包含若干个这样的控制流，它们各自拥有自己的运行栈和程序计数器，因而可以并发地执行。由于多个线程之间不可避免地存在同步问题，因此多线程编程要比一般的顺序编程复杂得多。

下面给出的是 Thread 类声明的头部(摘自 java. lang. Thread. java 1. 01 版)：pubic class Thread implements Runablle {

. . . . . .

我们注意到 Thread 类实现了一个名为 Runable 的接口，它也属于 java.lang 程序包。这个接口中只有一个名为 run 的方法，正是这个方法构成了线程的主体：一个线程被创建并出初化后，运行系统会调用它的 run()方法，线程因此获得运行。该方法一旦结束返回，也就意味着线程的死亡(当然，还可以用其它方法杀死一个线程，如调用其 stop())。从程序设计的角度来看，多线程编程的主要工作就是为线程编写 run()方法。具体地说可以有两种办法创建自己的线程：其一是创建一个 Thread 的子类，并重写其中的 run()方法；其二是自己声明一个实现 Runable 接口的类，为这个类编写 run()方法，然后以该类为"目标"(target)创建一个线程，此时新建线程的 run() 方法将仅仅包含一个对目标的 run()的调用。

线程同进程一样，有自己的生命周期(life circle)。一个线程在其生命周期中的不同阶段可能处于不同的状态，这些状态是：新线程态，可运行状态，死亡态。下面一一介绍：

·新线程态(new thread)：从使用 new 操作符创建一个线程开始一直到调用 start() 方法真正启动它，这个线程都处于所谓的"新线程态"。此时它仅仅是一个空的 Thread 对象而已，线程运行所需的系统资源尚未分配。处于这一状态的线程可以被启动(start)或终止(stop)。除此之外不能调用该对象的其它方法。

·可运行态(runable)：调用 Thread 的 start() 方法可为线程获取必要的系统资源，并导致 run()方法被调用，此时线程处于可运行态。需要指出的是处于这一状态的线程并不一定真的在运行中，这是因为多数计算机都只有一个处理器，要想同时运行所有处于可运行态的线程是不可能的。在这种情况下运行系统必需采用某种调度方案使多个线程可以共享处理器。我们可以把可运行态理解为某种"就绪状态"，在这种状态下的线程随时都可能处在运行中。

·不可运行态(not runnble)：线程可能由于下述原因进入不可运行态：

a. 其 suspend() 方法被调用，线程被"挂起"

b. 其 sleep() 方法被调用，线程"睡眠"一段时间

c. 线程调用了 wait()方法以等待某个条件发生改变

d. 线程阻塞于输入输出操作

因以上原因进入不可运行态的线程可通过与各自的原因相对应的途径重新进入可运行态，具体列出如下：

a. 被挂起线程的 resume()方法被调用

b. 线程睡眠时间已足够长

c. 线程所等待的条件已满足

d. 输入输出命令已完成

·死亡态：一个线程可以通过两种方式进入死亡态，一是"自然死亡"，即其 run()方法结束返回；二是被"杀死"，例如它的 stop()方法被调用。

Java 运行系统对线程的调度算法是所谓"固定优先级调度"。每一个 Java 线程有一个固定的优先级，任何时刻系统将挑选所有可运行线程中优先级最高的一个来运行。这一调度算法是抢先式的，也就是说，在任意时刻如果有一个优先级比所有其他可运行线程都高的线程进入可运行态，则新来者将立刻获得运行权。

当多个线程在系统内并发执行时，程序的行为将是复杂和难以控制的，线程间的同步就显得非常重要。Java 从最基本的 Object 类开始就提供了实现线程同步的机制，其具体表现就是以 notify()、notifyAll()、wait() 等为主的一组方法，下面简要介绍一下。

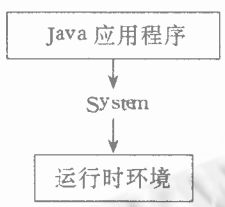
·wait()：该方法使当前线程(也就是此方法的调用者)进入不可运行态，以等待某种条件改变。线程将持续等待直到另外的线程调用当前对象的 notify()或 notifyAll() 方法为止。

·notify()/notifyAll()：唤醒一个或全部因调用当前对象的 wait()方法而进入等待状态的线程，通常该方法用于"通知"正在等待中的线程：条件已经满足，可以继续执行。

上面所列的方法都只能在具有 synchronized 修饰符的方法或语句块中使用，换言之，它们只能出现在临界区内，这是因为如果没有资源的互斥访问，线程同步也就无从谈起。

5. System

应用程序几乎总是要访问系统资源，如获取当前时间，执行标准输入输出操作等。按照以往的习惯，我们可以通过标准库例程甚至汇编代码来直接调用系统功能，但这样一来程序就失去了可移植性。每当把程序拿到一个新的运行环境时，就必需重写其中与系统相关的代码。Java 开发环境通过定义一个名为 System 的类解决了这个问题。System 为应用程序访问系统资源提供一个独立于系统的编程接口，使程序员从繁琐的细节中解脱出来。



System 所提供的系统资源主要包括以下几方面：

·标准输入流、输出流和出错信息流，它们分别对应 System 的三个成员变量：in、out 和 err。

·系统属性，如系统名称、用户名称、所使用的文件分隔符等。

. 垃圾收集

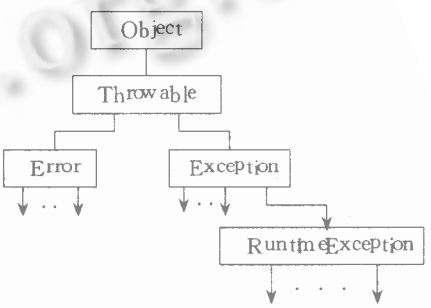
. 动态库装载

. 其他。包括数组拷贝，获取当前时间，退出运行环境等。

System 的所有变量和方法都是静态(static)的，我们无需自行创建一个 System 的实例(事实上 Java 也不允许这样做)，对其成员的访问只需冠以类名"System"。如："System.out.println("Hello!");"将在标准输出设备上输出"Hello!"。

6. Throwable 及其子类

在前一讲中已提到过 Java 用 Throwable 的不同子类来代表程序运行时可能遇到的各种异常(exception)情况，Java 的异常处理也是针对这一类对象进行。



. Error (错误)

当 Java 虚拟机中发生动态链接错误或其他"硬件"错误时就会产生一个 Error 型对象。一般情况下不必处理它。事实上，典型 Java 程序不大可能出现这种情况。

. Exception(异常)

程序运行期间可能出现的绝大多数异常情况都可以用 Exception 的某个子类来代表。它也是我们在程序中常常要处理的类型。需要特别提及的是它的一个子类—— RuntimeException

. RuntimeException(运行时异常)

这个类代表运行时在 Java 虚拟机内发生的异常。常见的有被零除、引用空指针等。这类异常不但频繁发生，而且程序员也难以预见到它们。考虑到这类异常的检测及处理代价很大，而且这样做本身也没有多少意义，Java 允许程序员对这类异常不予处理。下面的例子可以说明一般的 Exception 与 RuntimeException 的区别：假定在一段 Java 程序中需要对磁盘文件进行写入，我们很容易预见到可能因为写保护、磁盘满等原因而出现写操作失败，于是可以为其编写异常处理程序段；但我们却很难发觉程序的哪个部分在运行时可能引用空指针。后者是典型的 RuntimeException，而前者不是。在程序设计时也只考虑前一种情况的处理。

(待续)