InfoQ 中文站专栏合集 No.1

Java 深度历险

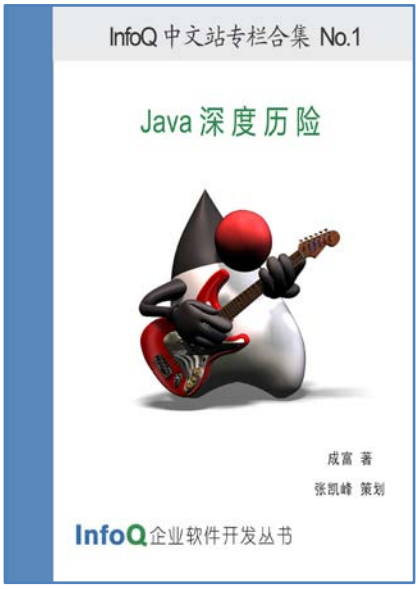
成富 著

张凯峰 策划

InfoQ企业软件开发丛书

免费在线版本

（非印刷免费在线版）



了解本书更多信息请登录本书的官方网站

InfoQ 中文站出品

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

http://www.infoq.com/cn/minibooks/java-explore

序

《Java 深度历险》专栏的作者成富，是 IBM 中国软件开发中心的高级工程师，也是我的前同事。他曾经是 CTO 毛新生的得意门生，承担过 Lotus Mashups 产品的重要研发职责，现在负责领导 Project Vulcan 项目的重要组件在中国团队的开发。成富对于 Java 和 Web 开发有着很深的造诣，同时在其他技术领域有着自己独到的见解。他是我见过的少有的具有极强技术领悟力和实践能力的一部分人之一。

成富还是一个专业的技术写手，看看他博客上的列表就知道，他在一年内会投递多少优质的稿件。所以顺理成章地，在我参与 InfoQ 中文站社区贡献时，很自然邀请他来开辟一个深入 Java 和 JVM 的专栏，他欣然应允，重要的是他以专业的技术作者素质，不再让我担心催稿，最终有了这十篇关于 Java 不同方面但深入浅出的主题内容。在几乎每篇专栏的结尾，都有多于平均数量的积极的评论，在 InfoQ 内部月度内容排行上尤为突出。同样是出于读者的呼声，才有了这本迷你书面世的可能。

很高兴地知道，成富接下来还会和华章有进一步的合作，撰写有关 Java 方面的技术书籍，让我们一起期待吧。

InfoQ 中文站原创团队主编　张凯峰

QCon全球企业开发大会（杭州站）

10月21日–10月23日 现在报名8折优惠！

大会主题

知名网站案例分析——阿里巴巴、淘宝、大众点评等知名网站背后的架构故事

脚本代码之美——专家解析HTML5、JavaScript、Node.JS设计中的难题

开放平台——来自百度、360、腾讯、盛大的案例分享

首席架构师的架构观——首席架构师眼中的简单原则

大数据和NoSQL——Hadoop、HBase、MongoDB和Cassandra等技术在当前的企业中的应用

DevOps——最前沿的开发&运维之道

运行中的云计算架构——云计算平台面面观，从架构到实践

Java依旧灿烂——Java使用者与平台架构师谈Java为何依旧灿烂

敏捷已到壮年——敏捷与精益开发的现状与未来

8月31日前报名8折优惠！

QCon

目录

InfoQ
Enterprise Software Development Community

时刻关注企业软件开发领域的变化与创新

我们的价值观：
创造可信赖的内容

我们的愿景：
帮助传播企业软件开发相关的知识和创新

我们的读者：
中高端技术人员，如架构师、项目经理、高级工程师等

我们的社区：
Java、.NET、Ruby、SOA、敏捷、架构和运维

我们的特质：
个性化RSS定制、国际化内容同步更新

我们的团队：
超过60位领域专家担当社区编辑

1

Java 字节代码的操纵

在一般的Java应用开发过程中，开发人员使用Java的方式比较简单。打开惯用的IDE，编写Java源代码，再利用IDE提供的功能直接运行Java 程序就可以了。这种开发模式背后的过程是：开发人员编写的是Java源代码文件（.java），IDE会负责调用Java的编译器把Java源代码编译成平台无关的字节代码（byte code），以类文件的形式保存在磁盘上（.class）。Java虚拟机（JVM）会负责把Java字节代码加载并执行。Java通过这种方式来实现其“编写一次，到处运行（Write once, run anywhere）”的目标。Java类文件中包含的字节代码可以被不同平台上的JVM所使用。Java字节代码不仅可以以文件形式存在于磁盘上，也可以通过网络方式来下载，还可以只存在于内存中。JVM中的类加载器会负责从包含字节代码的字节数组（byte[]）中定义出Java类。在某些情况下，可能会需要动态的生成 Java字节代码，或是对已有的Java字节代码进行修改。这个时候就需要用到本文中将要介绍的相关技术。首先介绍一下如何动态编译Java源文件。

动态编译 Java 源文件

在一般情况下，开发人员都是在程序运行之前就编写完成了全部的Java源代码并且成功编译。对有些应用来说，Java源代码的内容在运行时刻才能确定。这个时候就需要动态编译源代码来生成Java字节代码，再由JVM来加载执行。典型的场景是很多算法竞赛的在线评测系统（如 PKU JudgeOnline），允许用户上传Java代码，由系统在后台编译、运行并进行判定。在动态编译Java源文件时，使用的做法是直接在程序中调用Java编译器。

JSR 199引入了Java编译器API。如果使用JDK 6 的话，可以通过此API来动态编译Java代码。比如下面的代码用来动态编译最简单的Hello World类。该Java类的代码是保存在一个字符串中的。

```
public class CompilerTest {
   public static void main(String[] args) throws Exception {
      String source = "public class Main { public static void main(String[]
args) {System.out.println(\"Hello World!\");} }";
```

```
      JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
      StandardJavaFileManager fileManager =
         compiler.getStandardFileManager(null, null, null);
      StringSourceJavaObject sourceObject = new
         CompilerTest.StringSourceJavaObject("Main", source);
      Iterable< extends JavaFileObject> fileObjects =
         Arrays.asList(sourceObject);
      CompilationTask task = compiler.getTask(null, fileManager, null,
         null, null, fileObjects);
      boolean result = task.call();
      if (result) {
         System.out.println("编译成功。");
      }
   }

   static class StringSourceJavaObject extends SimpleJavaFileObject {

      private String content = null;
      public StringSourceJavaObject(String name, String content) throws
URISyntaxException {
         super(URI.create("string:///" + name.replace('.','/') +
            Kind.SOURCE.extension), Kind.SOURCE);
         this.content = content;
      }

      public CharSequence getCharContent(boolean
         ignoreEncodingErrors) throws IOException {
         return content;
      }
   }
}
```

如果不能使用JDK 6 提供的Java编译器API的话，可以使用JDK中的工具类 com.sun.tools.javac.Main，不过该工具类只能编译存放在磁盘上的文件，类似于直接使用javac命令。

另外一个可用的工具是 Eclipse JDT Core提供的编译器。这是Eclipse Java开发环境使用的增量式Java编译器 ,支持运行和调试有错误的代码。该编译器也可以单独使用。Play框架 在内部使用了JDT的编译器来动态编译Java源代码。在开发模式下，Play框架会定期扫描项目中的Java源代码文件，一旦发现有修改，会自动编译Java源代码。因此在修改代码之后，刷新页面就可以看到变化。使用这些动态编译的方式的时候，需要确保JDK中的tools.jar在应用的 CLASSPATH中。

下面介绍一个例子 ,是关于如何在Java里面做四则运算 ,比如求出来(3+4)*7-10 的值。

一般的做法是分析输入的运算表达式，自己来模拟计算过程。考虑到括号的存在和运算符的优先级等问题，这样的计算过程会比较复杂，而且容易出错。另外一种做法是可以用 JSR 223引入的脚本语言支持，直接把输入的表达式当做JavaScript或是JavaFX脚本来执行，得到结果。下面的代码使用的做法是动态生成Java源代码并编译，接着加载Java类来执行并获取结果。这种做法完全使用Java来实现。

```
private static double calculate(String expr) throws CalculationException
{
   String className = "CalculatorMain";
   String methodName = "calculate";
   String source = "public class " + className
      + " { public static double " + methodName + "() { return " + expr
+ "; } }";
      //省略动态编译Java源代码的相关代码，参见上一节
   boolean result = task.call();
   if (result) {
      ClassLoader loader = Calculator.class.getClassLoader();
      try {
         Class<?> clazz = loader.loadClass(className);
         Method method = clazz.getMethod(methodName, new Class<?>[] {});
         Object value = method.invoke(null, new Object[] {});
         return (Double) value;
      } catch (Exception e) {
         throw new CalculationException("内部错误。");
      }
   } else {
      throw new CalculationException("错误的表达式。");
   }
}
```

上面的代码给出了使用动态生成的 Java 字节代码的基本模式，即通过类加载器来加载字节代码，创建 Java 类的对象的实例，再通过 Java 反射 API 来调用对象中的方法。

Java 字节代码增强

Java 字节代码增强指的是在Java字节代码生成之后，对其进行修改，增强其功能。这种做法相当于对应用程序的二进制文件进行修改。在很多Java框架中都可以见到这种实现方式。Java字节代码增强通常与Java源文件中的注解（annotation）一块使用。注解在Java源代码中声明了需要增强的行为及 相关的元数据，由框架在运行时刻完成对字节代码的增强。Java字节代码增强应用的场景比较多，一般都集中在减少冗余代码和对开发人员屏蔽底层的实现细节 上。用过 JavaBeans的人可能对其中

那些必须添加的getter/setter方法感到很繁琐，并且难以维护。而通过字节代码增强，开发人员只需要声明Bean中的属性即可，getter/setter方法可以通过修改字节代码来自动添加。用过 JPA的人，在调试程序的时候，会发现 实体类 中被添加了一些额外的 域和方法。这些域和方法是在运行时刻由JPA的实现动态添加的。字节代码增强在 面向方面编程（AOP）的一些实现中也有使用。

在讨论如何进行字节代码增强之前，首先介绍一下表示一个 Java 类或接口的字节代码的组织形式。

> 类文件 {
>
> 0xCAFEBABE，小版本号，大版本号，常量池大小，常量池数组，
>
> 访问控制标记，当前类信息，父类信息，实现的接口个数，实现的接口信息数组，域个数，域信息数组，方法个数，方法信息数组，属性个数，属性信息数组
>
> }

如上所示，一个类或接口的字节代码使用的是一种松散的组织结构，其中所包含的内容依次排列。对于可能包含多个条目的内容，如所实现的接口、域、方法和属性等，是以数组来表示的。而在数组之前的是该数组中条目的个数。不同的内容类型，有其不同的内部结构。对于开发人员来说，直接操纵包含字节代码的字节数组的话，开发效率比较低，而且容易出错。已经有不少的开源库可以对字节代码进行修改或是从头开始创建新的Java类的字节代码内容。这些类库包括 ASM、cglib、serp和 BCEL 等。使用这些类库可以在一定程度上降低增强字节代码的复杂度。比如考虑下面一个简单的需求，在一个Java类的所有方法执行之前输出相应的日志。熟悉AOP的人都知道，可以用一个前增强（before advice）来解决这个问题。如果使用ASM的话，相关的代码如下：

```
ClassReader cr = new ClassReader(is);
ClassNode cn = new ClassNode();
cr.accept(cn, 0);
for (Object object : cn.methods) {
   MethodNode mn = (MethodNode) object;
   if ("<init>".equals(mn.name) || "<clinit>".equals(mn.name)) {
      continue;
   }
   InsnList insns = mn.instructions;
   InsnList il = new InsnList();
   il.add(new   FieldInsnNode(GETSTATIC,   "java/lang/System",   "out",
```

```
"Ljava/io/PrintStream;"));
   il.add(new LdcInsnNode("Enter method -> " + mn.name));
   il.add(new   MethodInsnNode(INVOKEVIRTUAL,   "java/io/PrintStream",
"println", "(Ljava/lang/String;)V"));
   insns.insert(il);  mn.maxStack += 3;
}
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

从 ClassWriter就 可以获取到包含增强之后的字节代码的字节数组 ,可以把字节代码写回磁盘或是由类加载器直接使用。上述示例中，增强部分的逻辑比较简单，只是遍历Java类中的所有方法并添加对System.out.println方法的调用。在字节代码中 ,Java方法体是由一系列的指令组成的。而要做的是生成调用 System.out.println方法的指令，并把这些指令插入到指令集合的最前面。ASM对这些指令做了抽象，不过熟悉全部的指令比较困难。ASM提供了一个工具类 ASMifierClassVisitor，可以打印出Java类的字节代码的结构信息。当需要增强某个类的时候 ,可以先在源代码上做出修改，再通过此工具类来比较修改前后的字节代码的差异，从而确定该如何编写增强的代码。

对类文件进行增强的时机是需要在 Java 源代码编译之后 ,在 JVM 执行之前。比较常见的做法有：

- 由IDE在完成编译操作之后执行。如Google App Engine的Eclipse插件会在编译之后运行 DataNucleus来对实体类进行增强。
- 在构建过程中完成，比如通过 Ant 或 Maven 来执行相关的操作。
- 实现自己的 Java 类加载器。当获取到 Java 类的字节代码之后 ,先进行增强处理，再从修改过的字节代码中定义出 Java 类。
- 通过 JDK 5 引入的 java.lang.instrument 包来完成。

java.lang.instrument

由于存在着大量对Java字节代码进行修改的需求，JDK 5引入了java.lang.instrument包并在 JDK 6中 得到了进一步的增强。基本的思路是在JVM启动的时候添加一些代理（agent）。每个代理是一个jar包，其清单（manifest）文件中会指定一个 代理类。这个类会包含一个premain方法。JVM在启动的时候会首先执行代理类的premain方法，再执行Java程序本身的main方法。在 premain方法中就可以对程序本身的字节

代码进行修改。JDK 6 中还允许在JVM启动之后动态添加代理。java.lang.instrument包支持两种修改的场景，一种是重定义一个Java类，即完全替换一个 Java类的字节代码；另外一种是转换已有的Java类，相当于前面提到的类字节代码增强。还是以前面提到的输出方法执行日志的场景为例，首先需要实现 java.lang.instrument.ClassFileTransformer接口来完成对已有Java类的转换。

```
static class MethodEntryTransformer implements ClassFileTransformer {
   public byte[] transform(ClassLoader loader, String className,
     Class<?> classBeingRedefined, ?ProtectionDomain protectionDomain,
byte[] classfileBuffer)
     throws  IllegalClassFormatException {
        try {
           ClassReader cr = new ClassReader(classfileBuffer);
           ClassNode cn = new ClassNode();
           //省略使用ASM进行字节代码转换的代码
           ClassWriter cw = new ClassWriter(0);
           cn.accept(cw);
           return cw.toByteArray();
        } catch (Exception e){
           return null;
        }
   }
}
```

有了这个转换类之后，就可以在代理的 premain 方法中使用它。

```
public static void premain(String args, Instrumentation inst) {
   inst.addTransformer(new MethodEntryTransformer());
}
```

把该代理类打成一个 jar 包，并在 jar 包的清单文件中通过 Premain-Class 声明代理类的名称。运行 Java 程序的时候，添加 JVM 启动参数-javaagent:myagent.jar。这样的话，JVM 会在加载 Java 类的字节代码之前，完成相关的转换操作。

总结

操纵 Java 字节代码是一件很有趣的事情。通过它，可以很容易的对二进制分发的 Java 程序进行修改，非常适合于性能分析、调试跟踪和日志记录等任务。另外一个非常重要的作用是把开发人员从繁琐的 Java 语法中解放出来。开发人员应该只需要负责编写与业务逻辑相关的重要代码。对于那些只是因为语法要求而添加的，或是模式固定的代码，完全可以将其字节代码动态生成出来。字节代码增强和源代码生成是不同的概念。源代码生成之后，就已经成为了程序的一部分，开发人员需要去维护

它：要么手工修改生成出来的源代码，要么重新生成。而字节代码的增强过程，对于开发人员是完全透明的。妥善使用 Java 字节代码的操纵技术，可以更好的解决某一类开发问题。

参考资料

- Java字节代码格式
- Java 6.0 Compiler API
- 深入探讨Java类加载器

2

Java 类的加载、链接和初始化

在 上一篇文章 中介绍了Java字节代码的操纵，其中提到了利用Java类加载器来加载修改过后的字节代码并在JVM上执行。本文接着上一篇的话题，讨论Java类的加载、链接和初始化。Java字节代码的表现形式是字节数组（byte[]），而Java类在JVM中的表现形式是 java.lang.Class类 的对象。一个Java类从字节代码到能够在JVM中被使用，需要经过加载、链接和初始化这三个步骤。这三个步骤中，对开发人员直接可见的是Java类的加 载，通过使用Java类加载器（class loader）可以在运行时刻动态的加载一个Java类；而链接和初始化则是在使用Java类之前会发生的动作。本文会详细介绍Java类的加载、链接和 初始化的过程。

Java 类的加载

Java类的加载是由类加载器来完成的。一般来说，类加载器分成两类：启动类加载器（bootstrap）和用户自定义的类加载器（user-defined）。两者的区别在于启动类加载器是由JVM的原生代码实现的，而用户自定义的类加载器都继承自Java中的 java.lang.ClassLoader类。在用户自定义类加载器的部分，一般JVM都会提供一些基本实现。应用程序的开发人员也可以根据需要编写自己的类加载器。JVM中最常使用的是系统类加载器（system），它用来启动Java应用程序的加载。通过java.lang.ClassLoader的 getSystemClassLoader()方法可以获取到该类加载器对象。

类加载器需要完成的最终功能是定义一个Java类，即把Java字节代码转换成JVM中的java.lang.Class类的对象。但是类加载的过程并不 是这么简单。Java类加载器有两个比较重要的特征：层次组织结构和代理模式。层次组织结构指的是每个类加载器都有一个父类加载器，通过 getParent()方法可以获取到。类加载器通过这种父亲-后代的方式组织在一起，形成树状层次结构。代理模式则指的是一个类加载器既可以自己完成Java类的定义工作，也可 以代理给其它的类加载器来完成。由于代理模式的存在，启动一个类的加载过程的类加载器和最终定义这个类的类加载器可能并不是一个。前者称为初始类加载器， 而后者称为定义类加载器。两者的关联在于：一个Java类的定义类加载器是该类所导入的其它Java类的初始类加载器。比如类A通过

import导入了类 B，那么由类A的定义类加载器负责启动类B的加载过程。

一般的类加载器在尝试自己去加载某个Java类之前，会首先代理给其父类加载器。当父类加载器找不到的时候，才会尝试自己加载。这个逻辑是封装在java.lang.ClassLoader类的 loadClass()方法中的。一般来说，父类优先的策略就足够好了。在某些情况下，可能需要采取相反的策略，即先尝试自己加载，找不到的时候再代理给父类加载器。这种做法在Java的Web容器中比较常见，也是 Servlet规范 推荐的做法。比如，Apache Tomcat为每个Web应用都提供一个独立的类加载器，使用的就是自己优先加载的策略。IBM WebSphere Application Server则允许Web应用选择类加载器使用的策略。

类加载器的一个重要用途是在JVM中为相同名称的Java类创建隔离空间。在JVM中，判断两个类是否相同，不仅是根据该类的 二进制名称 ，还需要根据两个类的定义类加载器。只有两者完全一样，才认为两个类的是相同的。因此，即便是同样的Java字节代码，被两个不同的类加载器定义之后，所得到的Java类也是不同的。如果试图在两个类的对象之间进行赋值操作，会抛出 java.lang.ClassCastException。这个特性为同样名称的Java类在JVM中共存创造了条件。在实际的应用中，可能会要求同一名称的Java类的不同版本在JVM中可以同时存在。通过类加载器就可以满足这种需求。这种技术在 OSGi中得到了广泛的应用。

Java 类的链接

Java类的链接指的是将Java类的二进制代码合并到JVM的运行状态之中的过程。在链接之前，这个类必须被成功加载。类的链接包括验证、准备和解析等几个步骤。验证是用来确保Java类的二进制表示在结构上是完全正确的。如果验证过程出现错误的话，会抛出 java.lang.VerifyError错误。准备过程则是创建Java类中的静态域，并将这些域的值设为默认值。准备过程并不会执行代码。在一个Java类中会包含对其它类或接口的形式引用，包括它的父类、所实现的接口、方法的形式参数和返回值的Java类等。解析的过程就是确保这些被引用的类能被正确的找到。解析的过程可能会导致其它的 Java类被加载。

不同的 JVM 实现可能选择不同的解析策略。一种做法是在链接的时候，就递归的把所有依赖的形式引用都进行解析。而另外的做法则可能是只在一个形式引用真正需要的时候才进行解析。也就是说如果一个 Java 类只是被引用了，但是并没有被真正用到，那么这个类有可能就不会被解析。考虑下面的代码：

```
public class LinkTest {
   public static void main(String[] args) {
      ToBeLinked toBeLinked = null;
      System.out.println("Test link.");
   }
}
```

类 LinkTest 引用了类 ToBeLinked，但是并没有真正使用它，只是声明了一个变量，并没有创建该类的实例或是访问其中的静态域。在 Oracle 的 JDK 6 中，如果把编译好的 ToBeLinked 的 Java 字节代码删除之后，再运行 LinkTest，程序不会抛出错误。这是因为 ToBeLinked 类没有被真正用到，而 Oracle 的 JDK 6 所采用的链接策略使得 ToBeLinked 类不会被加载，因此也不会发现 ToBeLinked 的 Java 字节代码实际上是不存在的。如果把代码改成 ToBeLinked toBeLinked = new ToBeLinked();之后，再按照相同的方法运行，就会抛出异常了。因为这个时候 ToBeLinked 这个类被真正使用到了，会需要加载这个类。

Java 类的初始化

当一个 Java 类第一次被真正使用到的时候，JVM 会进行该类的初始化操作。初始化过程的主要操作是执行静态代码块和初始化静态域。在一个类被初始化之前，它的直接父类也需要被初始化。但是，一个接口的初始化，不会引起其父接口的初始化。在初始化的时候，会按照源代码中从上到下的顺序依次执行静态代码块和初始化静态域。考虑下面的代码：

```
public class StaticTest {
   public static int X = 10;
   public static void main(String[] args) {
      System.out.println(Y); //输出60
   }
   static {
      X = 30;
   }
   public static int Y = X * 2;
}
```

在上面的代码中，在初始化的时候，静态域的初始化和静态代码块的执行会从上到下依次执行。因此变量 X 的值首先初始化成 10，后来又被赋值成 30；而变量 Y 的值则被初始化成 60。

Java 类和接口的初始化只有在特定的时机才会发生，这些时机包括：

- 创建一个 Java 类的实例。如

```
MyClass obj = new MyClass()
```

- 调用一个 Java 类中的静态方法。如

```
MyClass.sayHello()
```

- 给 Java 类或接口中声明的静态域赋值。如

```
MyClass.value = 10
```

- 访问 Java 类或接口中声明的静态域，并且该域不是常值变量。如

```
int value = MyClass.value
```

- 在顶层 Java 类中执行 assert 语句。

通过 Java 反射 API 也可能造成类和接口的初始化。需要注意的是，当访问一个 Java 类或接口中的静态域的时候，只有真正声明这个域的类或接口才会被初始化。考虑下面的代码：

```
class B {
   static int value = 100;
   static {
      System.out.println("Class B is initialized."); //输出
   }
}
class A extends B {
   static {
      System.out.println("Class A is initialized."); //不会输出
   }
}
public class InitTest {
   public static void main(String[] args) {
      System.out.println(A.value); //输出100
   }
}
```

在上述代码中，类 InitTest 通过 A.value 引用了类 B 中声明的静态域 value。由于 value 是在类 B 中声明的，只有类 B 会被初始化，而类 A 则不会被初始化。

创建自己的类加载器

在 Java 应用开发过程中，可能会需要创建应用自己的类加载器。典型的场景包括实现特定的 Java 字节代码查找方式、对字节代码进行加密/解密以及实现同名 Java 类的隔离等。创建自己的类加载器并不是一件复杂的事情，只需要继承自 java.lang.ClassLoader 类并覆写对应的方法即可。java.lang.ClassLoader 中提供的方法有不少，下面介绍几个创建类加载器时需要考虑的：

- defineClass() 这个方法用来完成从Java字节代码的字节数组到java.lang.Class的转换。这个方法是不能被覆写的，一般是用原生代码来实现的。
- findLoadedClass()：这个方法用来根据名称查找已经加载过的Java类。一个类加载器不会重复加载同一名称的类。
- findClass()：这个方法用来根据名称查找并加载Java类。
- loadClass()：这个方法用来根据名称加载Java类。
- resolveClass()：这个方法用来链接一个Java类。

这里比较 容易混淆的是 findClass()方法和 loadClass()方法的作用。前面提到过，在 Java 类的链接过程中，会需要对 Java 类进行解析，而解析可能会导致当前 Java 类所引用的其它 Java 类被加载。在这个时候，JVM 就是通过调用当前类的定义类加载器的 loadClass()方法来加载其它类的。findClass()方法则是应用创建的类加载器的扩展点。应用自己的类加载器应该覆写 findClass()方法来添加自定义的类加载逻辑。loadClass()方法的默认实现会负责调用 findClass()方法。

前面提到，类加载器的代理模式默认使用的是父类优先的策略。这个策略的实现是封装在 loadClass()方法中的。如果希望修改此策略，就需要覆写 loadClass()方法。

下面的代码给出了自定义的类加载的常见实现模式：

```
public class MyClassLoader extends ClassLoader {
   protected      Class<?>      findClass(String      name)      throws
ClassNotFoundException {
      byte[] b = null; //查找或生成Java类的字节代码
      return defineClass(name, b, 0, b.length);
   }
}
```

参考资料

- Java语言规范（第三版）- 第十三章：执行
- JVM规范（第二版） - 第五章：加载、链接和初始化
- 深入探讨Java类加载器

3

Java 线程：基本概念、可见性与同步

开发高性能并发应用不是一件容易的事情。这类应用的例子包括高性能Web服务器、游戏服务器和搜索引擎爬虫等。这样的应用可能需要同时处理成千上万个请求。对于这样的应用，一般采用多线程或事件驱动的架构。对于Java来说，在语言内部提供了线程的支持。但是Java的多线程应用开发会遇到很多问题。首先是很难编写正确，其次是很难测试是否正确，最后是出现问题时很难调试。一个多线程应用可能运行了好几天都没问题，然后突然就出现了问题，之后却又无法再次重现出来。如果在正确性之外，还需要考虑应用的吞吐量和性能优化的话，就会更加复杂。本文主要介绍Java中的线程的基本概念、可见性和线程同步相关的内容。

Java 线程基本概念

在操作系统中两个比较容易混淆的概念是进程（process）和线程（thread）。操作系统中的进程是资源的组织单位。进程有一个包含了程序内容和数据的地址空间，以及其它的资源，包括打开的文件、子进程和信号处理器等。不同进程的地址空间是互相隔离的。而线程表示的是程序的执行流程，是CPU调度的基本单位。线程有自己的程序计数器、寄存器、栈和帧等。引入线程的动机在于操作系统中阻塞式I/O的存在。当一个线程所执行的I/O被阻塞的时候，同一进程中的其它线程可以使用CPU来进行计算。这样的话，就提高了应用的执行效率。线程的概念在主流的操作系统和编程语言中都得到了支持。

一部分的Java程序是单线程的。程序的机器指令按照程序中给定的顺序依次执行。Java语言提供了java.lang.Thread类来为线程提供抽象。有两种方式创建一个新的线程：一种是继承java.lang.Thread类并覆写其中的run()方法，另外一种则是在创建java.lang.Thread类的对象的时候，在构造函数中提供一个实现了java.lang.Runnable接口的类的对象。在得到了java.lang.Thread类的对象之后，通过调用其start()方法就可以启动这个线程的执行。

一个线程被创建成功并启动之后，可以处在不同的状态中。这个线程可能正在占用

CPU 时间运行；也可能处在就绪状态，等待被调度执行；还可能阻塞在某个资源或是事件上。多个就绪状态的线程会竞争 CPU 时间以获得被执行的机会，而 CPU 则采用某种算法来调度线程的执行。不同线程的运行顺序是不确定的，多线程程序中的逻辑不能依赖于 CPU 的调度算法。

可见性

可见性（visibility）的问题是 Java 多线程应用中的错误的根源。在一个单线程程序中，如果首先改变一个变量的值，再读取该变量的值的时候，所读取到的值就是上次写操作写入的值。也就是说前面操作的结果对后面的操作是肯定可见的。但是在多线程程序中，如果不使用一定的同步机制，就不能保证一个线程所写入的值对另外一个线程是可见的。造成这种情况的原因可能有下面几个：

- CPU 内部的缓存：现在的 CPU 一般都拥有层次结构的几级缓存。CPU 直接操作的是缓存中的数据，并在需要的时候把缓存中的数据与主存进行同步。因此在某些时刻，缓存中的数据与主存内的数据可能是不一致的。某个线程所执行的写入操作的新值可能当前还保存在 CPU 的缓存中，还没有被写回到主存中。这个时候，另外一个线程的读取操作读取的就还是主存中的旧值。
- CPU 的指令执行顺序：在某些时候，CPU 可能改变指令的执行顺序。这有可能导致一个线程过早的看到另外一个线程的写入操作完成之后的新值。
- 编译器代码重排：出于性能优化的目的，编译器可能在编译的时候对生成的目标代码进行重新排列。

现实的情况是：不同的CPU可能采用不同的架构，而这样的问题在多核处理器和多处理器系统中变得尤其复杂。而Java的目标是要实现“编写一次，到处运行”，因此就有必要对Java程序访问和操作主存的方式做出规范，以保证同样的程序在不同的CPU架构上的运行结果是一致的。Java内存模型（Java Memory Model）就是为了这个目的而引入的。JSR 133则进一步修正了之前的内存模型中存在的问题。总得来说，Java内存模型描述了程序中共享变量的关系以及在主存中写入和读取这些变量值的底层细节。Java内存模型定义了Java语言中的 synchronized、volatile和 final等关键词对主存中变量读写操作的意义。Java开发人员使用这些关键词来描述程序所期望的行为，而编译器和JVM负责保证生成的代码在运行时刻的行为符合内存模型的描述。比如对声明为volatile的变量来说，在读取之前，JVM会确保CPU中缓存的值首先会失效，重新从主存中进行读取；而写入之后，新的值会被马上写入到主存中。而

synchronized和volatile关键词也会对编译器优化时候的代码重排带来额外的限制。比如编译器不能把synchronized块中的代码移出来。对volatile变量的读写操作是不能与其它读写操作一块重新排列的。

Java 内存模型中一个重要的概念是定义了“在之前发生（happens-before）”的顺序。如果一个动作按照“在之前发生”的顺序发生在另外一个动作之前，那么前一个动作的结果在多线程的情况下对于后一个动作就是肯定可见的。最常见的“在之前发生”的顺序包括：对一个对象上的监视器的解锁操作肯定发生在下一个对同一个监视器的加锁操作之前；对声明为 volatile 的变量的写操作肯定发生在后续的读操作之前。有了“在之前发生”顺序，多线程程序在运行时刻的行为在关键部分上就是可预测的了。编译器和 JVM 会确保“在之前发生”顺序可以得到保证。比如下面的一个简单的方法：

```
public void increase() {
    this.count++;
}
```

这是一个常见的计数器递增方法，this.count++实际是 this.count = this.count + 1，由一个对变量 this.count 的读取操作和写入操作组成。如果在多线程情况下，两个线程执行这两个操作的顺序是不可预期的。如果 this.count 的初始值是 1，两个线程可能都读到了为 1 的值，然后先后把 this.count 的值设为 2，从而产生错误。错误的原因在于其中一个线程对 this.count 的写入操作对另外一个线程是不可见的，另外一个线程不知道 this.count 的值已经发生了变化。如果在 increase() 方法声明中加上 synchronized 关键词，那就在两个线程的操作之间强制定义了一个“在之前发生”顺序。一个线程需要首先获得当前对象上的锁才能执行，在它拥有锁的这段时间完成对 this.count 的写入操作。而另一个线程只有在当前线程释放了锁之后才能执行。这样的话，就保证了两个线程对 increase()方法的调用只能依次完成，保证了线程之间操作上的可见性。

如果一个变量的值可能被多个线程读取，又能被最少一个线程锁写入，同时这些读写操作之间并没有定义好的“在之前发生”的顺序的话，那么在这个变量上就存在数据竞争（data race）。数据竞争的存在是 Java 多线程应用中要解决的首要问题。解决的办法就是通过 synchronized 和 volatile 关键词来定义好“在之前发生”顺序。

Java 中的锁

当数据竞争存在的时候，最简单的解决办法就是加锁。锁机制限制在同一时间只允

许一个线程访问产生竞争的数据的临界区。Java 语言中的 synchronized 关键字可以为一个代码块或是方法进行加锁。任何 Java 对象都有一个自己的监视器，可以进行加锁和解锁操作。当受到 synchronized 关键字保护的代码块或方法被执行的时候，就说明当前线程已经成功的获取了对象的监视器上的锁。当代码块或是方法正常执行完成或是发生异常退出的时候，当前线程所获取的锁会被自动释放。一个线程可以在一个 Java 对象上加多次锁。同时 JVM 保证了在获取锁之前和释放锁之后，变量的值是与主存中的内容同步的。

Java 线程的同步

在有些情况下，仅依靠线程之间对数据的互斥访问是不够的。有些线程之间存在协作关系，需要按照一定的协议来协同完成某项任务，比如典型的生产者-消费者模式。这种情况下就需要用到Java提供的线程之间的等待-通知机制。当线程所要求的条件不满足时，就进入等待状态；而另外的线程则负责在合适的时机发出通 知来唤醒等待中的线程。Java中的java.lang.Object类中的 wait/notify/notifyAll方法组就是完成线程之间的同步的。

在某个Java对象上面调用wait方法的时候，首先要检查当前线程是否获取到了这个对象上的锁。如果没有的话，就会直接抛出 java.lang.IllegalMonitorStateException异常。如果有锁的话，就把当前线程添加到对象的等待集合中，并释放其所拥有的锁。当前线程被阻塞，无法继续执行，直到被从对象的等待集合中移除。引起某个线程从对象的等待集合中移除的原因有很多：对象上的notify方法被调用时，该线程被选中；对象上的notifyAll方法被调用；线程被中断；对于有超时限制的wait操作，当超过时间限制时；JVM内部实现在非正常情况下的操作。

从上面的说明中，可以得到几条结论：wait/notify/notifyAll 操作需要放在 synchronized 代码块或方法中，这样才能保证在执行 wait/notify/notifyAll 的时候，当前线程已经获得了所需要的锁。当对于某个对象的等待集合中的线程数目没有把握的时候，最好使用 notifyAll 而不是 notify。notifyAll 虽然会导致线程在没有必要的情况下被唤醒而产生性能影响，但是在使用上更加简单一些。由于线程可能在非正常情况下被意外唤醒，一般需要把 wait 操作放在一个循环中，并检查所要求的逻辑条件是否满足。典型的使用模式如下所示：

```
private Object lock = new Object();
synchronized (lock) {
    while (/* 逻辑条件不满足的时候 */) {
```

```
        try {
            lock.wait();
        } catch (InterruptedException e) {}
    }
    //处理逻辑
}
```

上述代码中使用了一个私有对象 lock 来作为加锁的对象，其好处是可以避免其它代码错误的使用这个对象。

中断线程

通过一个线程对象的 interrupt()方 法可以向该线程发出一个中断请求。中断请求是一种线程之间的协作方式。当线程A通过调用线程B的interrupt()方法来发出中断请求的时候，线程A 是在请求线程B的注意。线程B应该在方便的时候来处理这个中断请求，当然这不是必须的。当中断发生的时候，线程对象中会有一个标记来记录当前的中断状态。通过 isInterrupted()方法可以判断是否有中断请求发生。如果当中断请求发生的时候，线程正处于阻塞状态，那么这个中断请求会导致该线程退出阻塞状态。可能造成线程处于阻塞状态的情况有：当线程通过调用wait()方法进入一个对象的等待集合中，或是通过 sleep()方法来暂时休眠，或是通过 join()方法来等待另外一个线程完成的时候。在线程阻塞的情况下，当中断发生的时候，会抛出 java.lang.InterruptedException， 代码会进入相应的异常处理逻辑之中。实际上在调用wait/sleep/join方法的时候，是必须捕获这个异常的。中断一个正在某个对象的等待集合中的线程，会使得这个线程从等待集合中被移除，使得它可以在再次获得锁之后，继续执行java.lang.InterruptedException异常的处 理逻辑。

通过中断线程可以实现可取消的任务。在任务的执行过程中可以定期检查当前线程的中断标记，如果线程收到了中断请求，那么就可以终止这个任务的执行。当遇到 java.lang.InterruptedException 的异常，不要捕获了之后不做任何处理。如果不想在这个层次上处理这个异常，就把异常重新抛出。当一个在阻塞状态的线程被中断并且抛出 java.lang.InterruptedException 异常的时候，其对象中的中断状态标记会被清空。如果捕获了 java.lang.InterruptedException 异常但是又不能重新抛出的话，需要通过再次调用 interrupt()方法来重新设置这个标记。

参考资料

- 现代操作系统

- Java语言规范（第三版）第 17 章：线程与锁
- Java内存模型FAQ
- Fixing the Java Memory Model, Part 1 & Part 2

4

Java 垃圾回收机制与引用类型

Java语言的一个重要特性是引入了自动的内存管理机制，使得开发人员不用自己来管理应用中的内存。C/C++开发人员需要通过 malloc/free 和 new/delete等函数来显式的分配和释放内存。这对开发人员提出了比较高的要求，容易造成内存访问错误和内存泄露等问题。一个常见的问题是会产生“悬挂引用 （ dangling references ）”，即一个对象引用所指向的内存区块已经被错误的回收并重新分配给新的对象了，程序如果继续使用这个引用的话会造成不可预期的结果。开发人员有可能忘记显式的调用释放内存的函数而造成内存泄露。而自动的内存管理则是把管理内存的任务交给编程语言的运行环境来完成。开发人员并不需要关心内存的分配和回收的底层细节。Java平台通过垃圾回收器来进行自动的内存管理。

Java 垃圾回收机制

Java 的垃圾回收器要负责完成 3 件任务：分配内存、确保被引用的对象的内存不被错误回收以及回收不再被引用的对象的内存空间。垃圾回收是一个复杂而且耗时的操作。如果 JVM 花费过多的时间在垃圾回收上，则势必会影响应用的运行性能。一般情况下，当垃圾回收器在进行回收操作的时候，整个应用的执行是被暂时中止（ stop-the-world ）的。这是因为垃圾回收器需要更新应用中所有对象引用的实际内存地址。不同的硬件平台所能支持的垃圾回收方式也不同。比如在多 CPU 的平台上，就可以通过并行的方式来回收垃圾。而单 CPU 平台则只能串行进行。不同的应用所期望的垃圾回收方式也会有所不同。服务器端应用可能希望在应用的整个运行时间中，花在垃圾回收上的时间总数越小越好。而对于与用户交互的应用来说，则可能希望所垃圾回收所带来的应用停顿的时间间隔越小越好。对于这种情况，JVM 中提供了多种垃圾回收方法以及对应的性能调优参数，应用可以根据需要来进行定制。

Java 垃圾回收机制最基本的做法是分代回收。内存中的区域被划分成不同的世代，对象根据其存活的时间被保存在对应世代的区域中。一般的实现是划分成 3 个世代：年轻、年老和永久。内存的分配是发生在年轻世代中的。当一个对象存活时间足够长的时候，它就会被复制到年老世代中。对于不同的世代可以使用不同的垃圾回收

算法。进行世代划分的出发点是对应用中对象存活时间进行研究之后得出的统计规律。一般来说，一个应用中的大部分对象的存活时间都很短。比如局部变量的存活时间就只在方法的执行过程中。基于这一点，对于年轻世代的垃圾回收算法就可以很有针对性。

年轻世代的内存区域被进一步划分成伊甸园（Eden）和两个存活区（survivor space）。伊甸园是进行内存分配的地方，是一块连续的空闲内存区域。在上面进行内存分配速度非常快，因为不需要进行可用内存块的查找。两个存活区中始终有一个是空白的。在进行垃圾回收的时候，伊甸园和其中一个非空存活区中还存活的对象根据其存活时间被复制到当前空白的存活区或年老世代中。经过这一次的复制之后，之前非空的存活区中包含了当前还存活的对象，而伊甸园和另一个存活区中的内容已经不再需要了，只需要简单地把这两个区域清空即可。下一次垃圾回收的时候，这两个存活区的角色就发生了交换。一般来说，年轻世代区域较小，而且大部分对象都已经不再存活，因此在其中查找存活对象的效率较高。

而对于年老和永久世代的内存区域，则采用的是不同的回收算法，称为“标记-清除-压缩（Mark-Sweep-Compact）”。标记的过程是找出当前还存活的对象，并进行标记；清除则遍历整个内存区域，找出其中需要进行回收的区域；而压缩则把存活对象的内存移动到整个内存区域的一端，使得另一端是一块连续的空闲区域，方便进行内存分配和复制。

JDK 5 中提供了 4 种不同的垃圾回收机制。最常用的是串行回收方式，即使用单个 CPU 回收年轻和年老世代的内存。在回收的过程中，应用程序被暂时中止。回收方式使用的是上面提到的最基本的分代回收。串行回收方式适合于一般的单 CPU 桌面平台。如果是多 CPU 的平台，则适合的是并行回收方式。这种方式在对年轻世代 进行回收的时候，会使用多个 CPU 来并行处理，可以提升回收的性能。并发标记-清除回收方式适合于对应用的响应时间要求比较 高的情况，即需要减少垃圾回收所带来的应用暂时中止的时间。这种做法的优点在于可以在应用运行的同时标记存活对象与回收垃圾，而只需要暂时中止应用比较短 的时间。

通过JDK中提供的 JConsole可以很容易的查看当前应用的内存使用情况。在JVM启动的时候添加参数 -verbose:gc 可以查看垃圾回收器的运行结果。

Java 引用类型

如果一个内存中的对象没有任何引用的话，就说明这个对象已经不再被使用了，从

而可以成为被垃圾回收的候选。不过由于垃圾回收器的运行时间不确定，可被垃圾回收的对象的实际被回收时间是不确定的。对于一个对象来说，只要有引用的存在，它就会一直存在于内存中。如果这样的对象越来越多，超出了JVM中的内存总数，JVM就会抛出 OutOfMemory错误。虽然垃圾回收的具体运行是由JVM来控制的，但是开发人员仍然可以在一定程度上与垃圾回收器进行交互，其目的在于更好的帮助垃圾回收器管理好应用的内存。这种交互方式就是使用JDK 1.2 引入的 java.lang.ref 包。

强引用

在一般的 Java 程序中，见到最多的就是强引用(strong reference)。如 Date date = new Date()，date 就是一个对象的强引用。对象的强引用可以在程序中到处传递。很多情况下，会同时有多个引用指向同一个对象。强引用的存在限制了对象在内存中的存活时间。假如对象 A 中包含了一个对象 B 的强引用，那么一般情况下，对象 B 的存活时间就不会短于对象 A。如果对象 A 没有显式的把对象 B 的引用设为 null 的话，就只有当对象 A 被垃圾回收之后，对象 B 才不再有引用指向它，才可能获得被垃圾回收的机会。

除了强引用之外，java.lang.ref 包中提供了对一个对象的不同的引用方式。JVM 的垃圾回收器对于不同类型的引用有不同的处理方式。

软引用

软引用（soft reference）在强度上弱于强引用，通过类 SoftReference来表示。它的作用是告诉垃圾回收器，程序中的哪些对象是不那么重要，当内存不足的时候是可以被暂时回收的。当JVM中的内存不足的时候，垃圾回收器会释放那 些只被软引用所指向的对象。如果全部释放完这些对象之后，内存还不足，才会抛出OutOfMemory错误。软引用非常适合于创建缓存。当系统内存不足的时候，缓存中的内容是可以被释放的。比如考虑一个图像编辑器的程序。该程序会把图像文件的全部内容都读取到内存中，以方便进行处理。而用户也可以同时打开 多个文件。当同时打开的文件过多的时候，就可能造成内存不足。如果使用软引用来指向图像文件内容的话，垃圾回收器就可以在必要的时候回收掉这些内存。

```
public class ImageData {
    private String path;
```

```
    private SoftReference<byte[]> dataRef;
    public ImageData(String path) {
        this.path = path;
        dataRef = new SoftReference<byte[]>(new byte[0]);
    }
    private byte[] readImage() {
        return new byte[1024 * 1024]; //省略了读取文件的操作
    }
    public byte[] getData() {
        byte[] dataArray = dataRef.get();
        if (dataArray == null || dataArray.length == 0) {
            dataArray = readImage();
            dataRef = new SoftReference<byte[]>(dataArray);
        }
        return dataArray;
    }
}
```

在运行上面程序的时候，可以使用 -Xmx 参数来限制JVM可用的内存。由于软引用所指向的对象可能被回收掉，在通过 get方法来获取软引用所实际指向的对象的时候，总是要检查该对象是否还存活。

弱引用

弱引用（weak reference）在强度上弱于软引用，通过类 WeakReference来 表示。它的作用是引用一个对象，但是并不阻止该对象被回收。如果使用一个强引用的话，只要该引用存在，那么被引用的对象是不能被回收的。弱引用则没有这个问题。在垃圾回收器运行的时候，如果一个对象的所有引用都是弱引用的话，该对象会被回收。弱引用的作用在于解决强引用所带来的对象之间在存活时间上的耦合关系。弱引用最常见的用处是在集合类中，尤其在哈希表中。哈希表的接口允许使用任何Java对象作为键来使用。当一个键值对被放入到哈希表中之后，哈希表 对象本身就有了对这些键和值对象的引用。如果这种引用是强引用的话，那么只要哈希表对象本身还存活，其中所包含的键和值对象是不会被回收的。如果某个存活 时间很长的哈希表中包含的键值对很多，最终就有可能消耗掉JVM中全部的内存。

对于这种情况的解决办法就是使用弱引用来引用这些对象，这样哈希表中的键和值对象都能被垃圾回收。Java中提供了 WeakHashMap来满足这一常见需求。

幽灵引用

在介绍幽灵引用之前，要先介绍Java提供的 对象终止化机制（finalization）。在Object类里面有个 finalize方法，其设计的初衷是在一个对象被真正回收之前，可以用来执行一些清理的工作。因为Java并没有提供类似C++的析构函数一样的机制，就通过finalize方法来实现。但是问题在于垃圾回收器的运行时间是不固定的，所以这些清理工作的实际运行时间也是不能预知的。幽灵引用（phantom reference）可以解决这个问题。在创建幽灵引用 PhantomReference的时候必须要指定一个引用队列。当一个对象的finalize方法已经被调用了之后，这个对象的幽灵引用会被加入到队列中。通过检查该队列里面的内容就知道一个对象是不是已经准备要被回收了。

幽灵引用及其队列的使用情况并不多见，主要用来实现比较精细的内存使用控制，这对于移动设备来说是很有意义的。程序可以在确定一个对象要被回收之后，再申请内存创建新的对象。通过这种方式可以使得程序所消耗的内存维持在一个相对较低的数量。比如下面的代码给出了一个缓冲区的实现示例。

```
public class PhantomBuffer {
  private byte[] data = new byte[0];
    private ReferenceQueue<byte[]> queue = new ReferenceQueue<byte[]>();
    private       PhantomReference<byte[]>       ref       =       new
PhantomReference<byte[]>(data, queue);
    public byte[] get(int size) {
        if (size <= 0) {
            throw new IllegalArgumentException("Wrong buffer size");
        }
        if (data.length < size) {
            data = null;
            System.gc(); //强制运行垃圾回收器
             try {
                queue.remove(); //该方法会阻塞直到队列非空
                ref.clear(); //幽灵引用不会自动清空，要手动运行
                ref = null;
                data = new byte[size];
                ref = new PhantomReference<byte[]>(data, queue);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
       }
       return data;
    }
}
```

在上面的代码中，每次申请新的缓冲区的时候，都首先确保之前的缓冲区的字节数组已经被成功回收。引用队列的 remove 方法会阻塞直到新的幽灵引用被加入到队列中。不过需要注意的是，这种做法会导致垃圾回收器被运行的次数过多，可能会造成程序的吞吐量过低。

引用队列

在有些情况下，程序会需要在一个对象的可达到性发生变化的时候得到通知。比如某个对象的强引用都已经不存在了，只剩下软引用或是弱引用。但是还需要对引用本身做一些的处理。典型的情景是在哈希表中。引用对象是作为WeakHashMap中的键对象的，当其引用的实际对象被垃圾回收之后，就需要把该键值对从哈希表中删除。有了引用队列（ReferenceQueue），就可以方便的获取到这些弱引用对象，将它们从表中删除。在软引用和弱引用对象被添加到队列之前，其对实际对象的引用会被自动清空。通过引用队列的 poll/remove方法就可以分别以非阻塞和阻塞的方式获取队列中的引用对象。

参考资料

- Java内存管理白皮书
- Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine
- Plugging memory leaks with soft references
- Plugging memory leaks with weak references

5

Java 泛型

Java泛型（generics）是JDK 5 中引入的一个新特性，允许在定义类和接口的时候使用类型参数（type parameter）。声明的类型参数在使用时用具体的类型来替换。泛型最主要的应用是在JDK 5 中的新 集合类框架 中。对于泛型概念的引入，开发社区的观点是 褒贬不一 。 从好的方面来说，泛型的引入可以解决之前的集合类框架在使用过程中通常会出现的运行时刻类型错误，因为编译器可以在编译时刻就发现很多明显的错误。而从不好的地方来说，为了保证与旧有版本的兼容性，Java泛型的实现上存在着一些不够优雅的地方。当然这也是任何有历史的编程语言所需要承担的历史包袱。后续的版本更新会为早期的设计缺陷所累。

开发人员在使用泛型的时候，很容易根据自己的直觉而犯一些错误。比如一个方法如果接收 List<Object>作为形式参数，那么如果尝试将一个 List<String>的对象作为实际参数传进去，却发现无法通过编译。虽然从直觉上来说，Object 是 String 的父类，这种类型转换应该是合理的。但是实际上这会产生隐含的类型转换问题，因此编译器直接就禁止这样的行为。本文试图对 Java 泛型做一个概括性的说明。

类型擦除

正确理解泛型概念的首要前提是理解类型擦除（type erasure）。 Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。这个过程就称为类型擦除。如在代码中定义的List<Object>和List<String>等类型，在编译之后都会变成List。JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的。Java编译器会在编译时尽可能的发现可能出错的 地方，但是仍然无法避免在运行时刻出现类型转换异常的情况。类型擦除也是Java的泛型实现方式与 C++模板机制 实现方式之间的重要区别。

很多泛型的奇怪特性都与这个类型擦除的存在有关，包括：

- 泛型类并没有自己独有的 Class 类对象。比如并不存在 List<String>.class 或是

List<Integer>.class，而只有 List.class。

- 静态变量是被泛型类的所有实例所共享的。对于声明为 MyClass<T>的类，访问其中的静态变量的方法仍然是 MyClass.myStaticVar。不管是通过 new MyClass<String>还是 new MyClass<Integer>创建的对象，都是共享一个静态变量。
- 泛型的类型参数不能用在 Java 异常处理的 catch 语句中。因为异常处理是由 JVM 在运行时刻来进行的。由于类型信息被擦除，JVM 是无法区分两个异常类型 MyException<String>和 MyException<Integer>的。对于 JVM 来说，它们都是 MyException 类型的。也就无法执行与异常对应的 catch 语句。

类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是 Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。同时去掉出现的类型声明，即去掉<>的内容。比如 T get()方法声明就变成了 Object get()；List<String>就变成了 List。接下来就可能需要生成一些桥接方法（bridge method）。这是由于擦除了类型之后的类可能缺少某些必须的方法。比如考虑下面的代码：

```
class MyString implements Comparable<String> {
    public int compareTo(String str) {
        return 0;
    }
}
```

当类型信息被擦除之后，上述类的声明变成了 class MyString implements Comparable。但是这样的话，类 MyString 就会有编译错误，因为没有实现接口 Comparable 声明的 int compareTo(Object)方法。这个时候就由编译器来动态生成这个方法。

实例分析

了解了类型擦除机制之后，就会明白编译器承担了全部的类型检查工作。编译器禁止某些泛型的使用方式，正是为了确保类型的安全性。以上面提到的 List<Object>和 List<String>为例来具体分析：

```
public void inspect(List<Object> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
    list.add(1); //这个操作在当前方法的上下文是合法的。
```

```
}
public void test() {
    List<String> strs = new ArrayList<String>();
    inspect(strs); //编译错误
}
```

这段代码中，inspect方法接受List<Object>作为参数，当在test方法中试图传入List<String>的 时候，会出现编译错误。假设这样的做法是允许的，那么在inspect方法就可以通过list.add(1)来向集合中添加一个数字。这样在test方法看来，其声明为List<String>的集合中却被添加了一个Integer类型的对象。这显然是违反类型安全的原则的，在某个时候肯定会 抛出 ClassCastException。因此，编译器禁止这样的行为。编译器会尽可能的检查可能存在的类型安全问题。对于确定是违反相关原则的地方，会给出编译错误。当编译器无法判断类型的使用是否正确的时候，会给出警告信息。

通配符与上下界

在使用泛型类的时候，既可以指定一个具体的类型，如 List<String>就声明了具体的类型是 String；也可以用通配符?来表示未知类型，如 List<?>就声明了 List 中包含的元素类型是未知的。 通配符所代表的其实是一组类型，但具体的类型是未知的。List<?>所声明的就是所有类型都是可以的。但是 List<?>并不等同于 List<Object>。List<Object>实际上确定了 List 中包含的是 Object 及其子类，在使用的时候都可以通过 Object 来进行引用。而 List<?>则其中所包含的元素类型是不确定。其中可能包含的是 String，也可能是 Integer。如果它包含了 String 的话，往里面添加 Integer 类型的元素就是错误的。正因为类型未知，就不能通过 new ArrayList<?>()的方法来创建一个新的 ArrayList 对象。因为编译器无法知道具体的类型是什么。但是对于 List<?>中的元素确总是可以用 Object 来引用的，因为虽然类型未知，但肯定是 Object 及其子类。考虑下面的代码：

```
public void wildcard(List<?> list) {
    list.add(1);//编译错误
}
```

如上所示，试图对一个带通配符的泛型类进行操作的时候，总是会出现编译错误。其原因在于通配符所表示的类型是未知的。

因为对于 List<?>中的元素只能用 Object 来引用，在有些情况下不是很方便。在这些情况下，可以使用上下界来限制未知类型的范围。如 List<? extends Number>说明 List 中可能包含的元素类型是 Number 及其子类。而 List<? super Number>则说明 List 中

包含的是 Number 及其父类。当引入了上界之后，在使用类型的时候就可以使用上界类中定义的方法。比如访问 List<? extends Number>的时候，就可以使用 Number 类的 intValue 等方法。

类型系统

在Java中，大家比较熟悉的是通过继承机制而产生的类型体系结构。比如String继承自Object。根据 Liskov替换原则 ，子类是可以替换父类的。当需要Object类的引用的时候，如果传入一个String对象是没有任何问题的。但是反过来的话，即用父类的引用替换子类引用 的时候，就需要进行强制类型转换。编译器并不能保证运行时刻这种转换一定是合法的。这种自动的子类替换父类的类型转换机制，对于数组也是适用的。String[]可以替换Object[]。但是泛型的引入，对于这个类型系统产生了一定的影响。正如前面提到的List<String>是 不能替换掉List<Object>的。

引入泛型之后的类型系统增加了两个维度：一个是类型参数自身的继承体系结构，另外一个是泛型类或接口自身的继承体系结构。第一个指的是对于 List<String>和 List<Object>这样的情况，类型参数 String 是继承自 Object 的。而第二种指的是 List 接口继承自 Collection 接口。对于这个类型系统，有如下的一些规则：

- 相同类型参数的泛型类的关系取决于泛型类自身的继承体系结构。即 List<String> 是 Collection<String> 的子类型，List<String> 可以替换 Collection<String>。这种情况也适用于带有上下界的类型声明。
- 当泛型类的类型声明中使用了通配符的时候，其子类型可以在两个维度上分别展开。如对 Collection<? extends Number>来说，其子类型可以在 Collection 这个维度上展开，即 List<? extends Number>和 Set<? extends Number>等；也可以在 Number 这个层次上展开，即 Collection<Double>和 Collection<Integer>等。如此循环下去，ArrayList<Long>和 HashSet<Double>等也都算是 Collection<? extends Number>的子类型。
- 如果泛型类中包含多个类型参数，则对于每个类型参数分别应用上面的规则。

理解了上面的规则之后，就可以很容易的修正实例分析中给出的代码了。只需要把 List<Object>改成 List<?>即可。List<String>是 List<?>的子类型，因此传递参数时不会发生错误。

开发自己的泛型类

泛型类与一般的 Java 类基本相同，只是在类和接口定义上多出来了用<>声明的类型参数。一个类可以有多个类型参数，如 MyClass<X, Y, Z>。 每个类型参数在声明的时候可以指定上界。所声明的类型参数在 Java 类中可以像一般的类型一样作为方法的参数和返回值，或是作为域和局部变量的类型。但是由于类型擦除机制，类型参数并不能用来创建对象或是作为静态变量的类型。考虑下面的泛型类中的正确和错误的用法。

```
class ClassTest<X extends Number, Y, Z> {
    private X x;
    private static Y y; //编译错误，不能用在静态变量中
    public X getFirst() {
        //正确用法
        return x;
    }
    public void wrong() {
        Z z = new Z(); //编译错误，不能创建对象
    }
}
```

最佳实践

在使用泛型的时候可以遵循一些基本的原则，从而避免一些常见的问题。

- 在代码中避免泛型类和原始类型的混用。比如 List<String>和 List 不应该共同使用。这样会产生一些编译器警告和潜在的运行时异常。当需要利用 JDK 5 之前开发的遗留代码，而不得不这么做时，也尽可能的隔离相关的代码。
- 在使用带通配符的泛型类的时候，需要明确通配符所代表的一组类型的概念。由于具体的类型是未知的，很多操作是不允许的。
- 泛型类最好不要同数组一块使用。你只能创建 new List<?>[10]这样的数组，无法创建 new List<String>[10]这样的。这限制了数组的使用能力，而且会带来很多费解的问题。因此，当需要类似数组的功能时候，使用集合类即可。
- 不要忽视编译器给出的警告信息。

参考资料

- Generics gotchas
- Java Generics FAQs
- Generics in Java Programming Language

6

Java 注解

在开发Java程序，尤其是Java EE应用的时候，总是免不了与各种配置文件打交道。以Java EE中典型的S(pring)S(truts)H(ibernate)架构来说，Spring、Struts和 Hibernate这三个框架都有自己的XML格式的配置文件。这些配置文件需要与Java源代码保存同步，否则的话就可能出现错误。而且这些错误有可能到了运行时刻才被发现。把同一份信息保存在两个地方，总是个坏的主意。理想的情况是在一个地方维护这些信息就好了。其它部分所需的信息则通过自动的方式来生成。JDK 5 中引入了源代码中的注解（annotation）这一机制。注解使得Java源代码中不但可以包含功能性的实现代码，还可以添加元数据。注解的功能类似于代码中的注释，所不同的是注解不是提供代码功能的说明，而是实现程序功能的重要组成部分。Java注解已经在很多框架中得到了广泛的使用，用来简化程序中的配置。

使用注解

在一般的Java开发中，最常接触到的可能就是 @Override和 @SupressWarnings这 两个注解了。使用@Override的时候只需要一个简单的声明即可。这种称为标记注解（marker annotation），它的出现就代表了某种配置语义。而其它的注解是可以有自己的配置参数的。配置参数以名值对的方式出现。使用@SupressWarnings的时候需要类似@SupressWarnings({"uncheck", "unused"})这样的语法。在括号里面的是该注解可供配置的值。由于这个注解只有一个配置参数，该参数的名称默认为value，并且可以省略。而花括号则表示是数组类型。在 JPA中的 @Table注解使用类似@Table(name = "Customer", schema = "APP")这样的语法。从这里可以看到名值对的用法。在使用注解时候的配置参数的值必须是编译时刻的常量。

从某种角度来说，可以把注解看成是一个 XML 元素，该元素可以有不同的预定义的属性。而属性的值是可以在声明该元素的时候自行指定的。在代码中使用注解，就相当于把一部分元数据从 XML 文件移到了代码本身之中，在一个地方管理和维护。

开发注解

在一般的开发中，只需要通过阅读相关的 API 文档来了解每个注解的配置参数的含义，并在代码中正确使用即可。在有些情况下，可能会需要开发自己的注解。这在库的开发中比较常见。注解的定义有点类似接口。下面的代码给出了一个简单的描述代码分工安排的注解。通过该注解可以在源代码中记录每个类或接口的分工和进度情况。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Assignment {
    String assignee();
    int effort();
    double finished() default 0;
}
```

@interface用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型。可以通过default来声明参数的默认值。在这里可以看到 @Retention和 @Target这样的元注解，用来声明注解本身的行为。@Retention用来声明注解的保留策略，有 CLASS、RUNTIME和 SOURCE这三种，分别表示注解保存在类文件、JVM运行时刻和源代码中。只有当声明为RUNTIME的时候，才能够在运行时刻通过反射API来获取到注解的信息。@Target用来声明注解可以被添加在哪些类型的元素上，如类型、方法和域等。

处理注解

在程序中添加的注解，可以在编译时刻或是运行时刻来进行处理。在编译时刻处理的时候，是分成多趟来进行的。如果在某趟处理中产生了新的Java源文件，那么就需要另外一趟处理来处理新生成的源文件。如此往复，直到没有新文件被生成为止。在完成处理之后，再对Java代码进行编译。JDK 5 中提供了 apt工具用来对注解进行处理。apt是一个命令行工具，与之配套的还有一套用来描述程序语义结构的 Mirror API。Mirror API(com.sun.mirror.*)描述的是程序在编译时刻的静态结构。通过Mirror API可以获取到被注解的Java类型元素的信息，从而提供相应的处理逻辑。具体的处理工作交给apt工具来完成。编写注解处理器的核心是 AnnotationProcessorFactory 和 AnnotationProcessor两个接口。后者表示的是注解处理器，而前者则是为某些注解类型创建注解处理器的工厂。

以上面的注解 Assignment 为例，当每个开发人员都在源代码中更新进度的话，就可

以通过一个注解处理器来生成一个项目整体进度的报告。首先是注解处理器工厂的实现。

```
public class AssignmentApf implements AnnotationProcessorFactory {
    public AnnotationProcessor
getProcessorFor(Set<AnnotationTypeDeclaration> atds,?
AnnotationProcessorEnvironment env) {
        if (atds.isEmpty()) {
            return AnnotationProcessors.NO_OP;
        }
        return new AssignmentAp(env); //返回注解处理器
    }
    public Collection<String> supportedAnnotationTypes() {
        return
Collections.unmodifiableList(Arrays.asList("annotation.Assignment"))
;
    }
    public Collection<String> supportedOptions() {
        return Collections.emptySet();
    }
}
```

AnnotationProcessorFactory接口有三个方法：getProcessorFor是根据注解的类型来返回特定的注解处理 器；supportedAnnotationTypes是返回该工厂生成的注解处理器所能支持的注解类型；supportedOptions用来表示所支 持的附加选项。在运行apt命令行工具的时候，可以通过-A来传递额外的参数给注解处理器，如-Averbose=true。当工厂通过 supportedOptions方法声明了所能识别的附加选项之后，注解处理器就可以在运行时刻通过 AnnotationProcessorEnvironment的getOptions方法获取到选项的实际值。注解处理器本身的基本实现如下所示。

```
public class AssignmentAp implements AnnotationProcessor {
    private AnnotationProcessorEnvironment env;
    private AnnotationTypeDeclaration assignmentDeclaration;
    public AssignmentAp(AnnotationProcessorEnvironment env) {
        this.env = env;
        assignmentDeclaration = (AnnotationTypeDeclaration)
env.getTypeDeclaration("annotation.Assignment");
    }
    public void process() {
        Collection<Declaration> declarations =
env.getDeclarationsAnnotatedWith(assignmentDeclaration);
        for (Declaration declaration : declarations) {
            processAssignmentAnnotations(declaration);
        }
    }
    private void processAssignmentAnnotations(Declaration declaration)
```

```
{
       Collection<AnnotationMirror>           annotations           =
declaration.getAnnotationMirrors();
       for (AnnotationMirror mirror : annotations) {
          if
(mirror.getAnnotationType().getDeclaration().equals(assignmentDeclar
ation)) {
              Map<AnnotationTypeElementDeclaration,   AnnotationValue>
values = mirror.getElementValues();
              String assignee = (String) getAnnotationValue(values,
                                  "assignee"); //获取注解的值
          }
       }
   }
}
```

注解处理器的处理逻辑都在process方法中完成。通过一个声明（Declaration）的getAnnotationMirrors方法就可以获取到该声明上所添加的注解的实际值。得到这些值之后，处理起来就不难了。

在创建好注解处理器之后，就可以通过 apt 命令行工具来对源代码中的注解进行处理。 命令的运行格式是 apt -classpath bin -factory annotation.apt.AssignmentApf src/annotation/work/*.java，即通过-factory 来指定注解处理器工厂类的名称。实际上，apt 工具在完成处理之后，会自 动调用 javac 来编译处理完成后的源代码。

JDK 5 中的apt工具的不足之处在于它是Oracle提供的私有实现。在JDK 6 中，通过 JSR 269 把自定义注解处理器这一功能进行了规范化，有了新的 javax.annotation.processing这个新的API。对Mirror API也进行了更新，形成了新的 javax.lang.model包。注解处理器的使用也进行了简化，不需要再单独运行apt这样的命令行工具，Java编译器本身就可以完成对注解的处理。对于同样的功能，如果用JSR 269 的做法，只需要一个类就可以了。

```
@SupportedSourceVersion(SourceVersion.RELEASE_6)
@SupportedAnnotationTypes("annotation.Assignment")
public class AssignmentProcess extends AbstractProcessor {
   private TypeElement assignmentElement;
   public synchronized void init(ProcessingEnvironment processingEnv)
{
       super.init(processingEnv);
       Elements elementUtils = processingEnv.getElementUtils();
       assignmentElement =
            elementUtils.getTypeElement("annotation.Assignment");
   }
   public boolean process(Set<? extends TypeElement> annotations,
```

```
                RoundEnvironment roundEnv) {
        Set<? extends Element> elements =
                roundEnv.getElementsAnnotatedWith(assignmentElement);
        for (Element element : elements) {
            processAssignment(element);
        }
    }
    private void processAssignment(Element element) {
        List<?     extends     AnnotationMirror>     annotations     =
element.getAnnotationMirrors();
        for (AnnotationMirror mirror : annotations) {
            if
(mirror.getAnnotationType().asElement().equals(assignmentElement)) {
                Map<?     extends     ExecutableElement,     ?     extends
AnnotationValue>
                    values = mirror.getElementValues();
                String assignee = (String) getAnnotationValue(values,
                    "assignee"); //获取注解的值
            }
        }
    }
}
```

仔细比较上面两段代码，可以发现它们的基本结构是类似的。不同之处在于JDK 6中通过元注解 @SupportedAnnotationTypes来声明所支持的注解类型。另外描述程序静态结构的javax.lang.model包使用了不同的类型名称。使用的时候也更加简单，只需要通过javac -processor annotation.pap.AssignmentProcess Demo1.java这样的方式即可。

上面介绍的这两种做法都是在编译时刻进行处理的。而有些时候则需要在运行时刻来完成对注解的处理。这个时候就需要用到Java的反射API。反射 API提供了在运行时刻读取注解信息的支持。不过前提是注解的保留策略声明的是运行时。Java反射API的 AnnotatedElement接口提供了获取类、方法和域上的注解的实用方法。比如获取到一个Class类对象之后，通过getAnnotation方法就可以获取到该类上添加的指定注解类型的注解。

实例分析

下面通过一个具体的实例来分析说明在实践中如何来使用和处理注解。假定有一个公司的雇员信息系统，从访问控制的角度出发，对雇员的工资的更新只能由具有特定角色的用户才能完成。考虑到访问控制需求的普遍性，可以定义一个注解来让开

发人员方便的在代码中声明访问控制权限。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RequiredRoles {
    String[] value();
}
```

下一步则是如何对注解进行处理，这里使用的Java的反射API并结合 动态代理 。下面是动态代理中的InvocationHandler接口的实现。

```
public class AccessInvocationHandler<T> implements InvocationHandler {
    final T accessObj;
    public AccessInvocationHandler(T accessObj) {
        this.accessObj = accessObj;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        RequiredRoles                    annotation                    =
method.getAnnotation(RequiredRoles.class); //通过反射API获取注解
        if (annotation != null) {
            String[] roles = annotation.value();
            String role = AccessControl.getCurrentRole();
            if (!Arrays.asList(roles).contains(role)) {
                throw new AccessControlException("The user is not allowed
to invoke this method.");
            }
        }
        return method.invoke(accessObj, args);
    }
}
```

在具体使用的时候，首先要通过 Proxy.newProxyInstance 方法创建一个 EmployeeGateway 的接口的代理类，使用该代理类来完成实际的操作。

参考资料

- JDK 5和 JDK 6中的apt工具说明文档
- Pluggable Annotation Processing API
- APT: Compile-Time Annotation Processing with Java

7

Java 反射与动态代理

在上一篇文章中介绍Java注解的时候 多次提到了Java的反射API。与javax.lang.model不同的是，通过反射API可以获取程序在运行时刻的内部结构。反射API中提供的动态代理也是非常强大的功能，可以原生实现 AOP中的方法拦截功能。正如英文单词reflection的含义一样 ,使用反射API的时候就好像在看一个Java类在水中的倒影一样。知道了Java类的内部结构之后，就可以与它进行交互，包括创建新的对象和调用对象中的方法等。这种交互方式与直接在源代码中使用的效果是相同的，但是又额外提供了运行时刻的灵活性。使用反射的一个最大的弊端是 性能比较差 。相同的操作，用反射API所需的时间大概比直接的使用要慢一两个数量级。不过现在的JVM实现中，反射操作的性能已经有了 很大的提升 。在灵活性与性能之间，总是需要进行权衡的。应用可以在适当的时机来使用反射API。

基本用法

Java反射API的第一个主要作用是获取程序在运行时刻的内部结构。这对于程序的检查工具和调试器来说，是非常实用的功能。只需要短短的十几行代码，就可以遍历出来一个Java类的内部结构，包括其中的构造方法、声明的域和定义的方法等。这不得不说是一个很强大的能力。只要有了java.lang.Class类 的对象，就可以通过其中的方法来获取到该类中的构造方法、域和方法。对应的方法分别是 getConstructor、getField和 getMethod。这三个方法还有相应的getDeclaredXXX版本，区别在于getDeclaredXXX版本的方法只会获取该类自身所声明的元素，而不会考虑继承下来的。Constructor、Field和 Method这三个类分别表示类中的构造方法、域和方法。这些类中的方法可以获取到所对应结构的元数据。

反射 API 的另外一个作用是在运行时刻对一个 Java 对象进行操作。这些操作包括动态创建一个 Java 类的对象，获取某个域的值以及调用某个方法。在 Java 源代码中编写的对类和对象的操作，都可以在运行时刻通过反射 API 来实现。考虑下面一个简单的 Java 类。

```
class MyClass {
```

```
    public int count;
    public MyClass(int start) {
        count = start;
    }
    public void increase(int step) {
        count = count + step;
    }
}
```

使用一般做法和反射 API 都非常简单。

```
MyClass myClass = new MyClass(0); //一般做法
myClass.increase(2);
System.out.println("Normal -> " + myClass.count);
try {
    //获取构造方法
    Constructor constructor = MyClass.class.getConstructor(int.class);
    //创建对象
    MyClass myClassReflect = constructor.newInstance(10);
    //获取方法
    Method method = MyClass.class.getMethod("increase", int.class);
    //调用方法
    method.invoke(myClassReflect, 5);
    //获取域
    Field field = MyClass.class.getField("count");
    //获取域的值
    System.out.println("Reflect -> " + field.getInt(myClassReflect));
} catch (Exception e) {
    e.printStackTrace();
}
```

由于数组的特殊性，Array类提供了一系列的静态方法用来创建数组和对数组中的元素进行访问和操作。

```
Object array = Array.newInstance(String.class, 10); //等价于 new String[10]
Array.set(array, 0, "Hello"); //等价于array[0] = "Hello"
Array.set(array, 1, "World"); //等价于array[1] = "World"
System.out.println(Array.get(array, 0)); //等价于array[0]
```

使用Java反射API的时候可以绕过Java默认的访问控制检查，比如可以直接获取到对象的私有域的值或是调用私有方法。只需要在获取到Constructor、Field和Method类的对象之后，调用 setAccessible方法并设为true即可。有了这种机制，就可以很方便

的在运行时刻获取到程序的内部状态。

处理泛型

Java 5 中引入了泛型的概念之后，Java反射API也做了相应的修改，以提供对泛型的支持。由于类型擦除机制的存在，泛型类中的类型参数等信息，在运行时刻是不存在的。JVM看到的都是原始类型。对此，Java 5 对Java类文件的格式做了修订，添加了Signature属性，用来包含不在JVM类型系统中的类型信息。比如以java.util.List接口为例，在其类文件中的 Signature 属性的声明是<E:Ljava/lang/Object;>Ljava/lang/Object;Ljava/util /Collection<TE;>;;，这就说明List接口有一个类型参数E。在运行时刻，JVM会读取Signature属性的内容并提供给反射API来使用。

比如在代码中声明了一个域是 List<String>类型的，虽然在运行时刻其类型会变成原始类型 List，但是仍然可以通过反射来获取到所用的实际的类型参数。

```
Field field = Pair.class.getDeclaredField("myList"); //myList的类型是
List
Type type = field.getGenericType();
if (type instanceof ParameterizedType) {
    ParameterizedType paramType = (ParameterizedType) type;
    Type[] actualTypes = paramType.getActualTypeArguments();
    for (Type aType : actualTypes) {
        if (aType instanceof Class) {
            Class clz = (Class) aType;
            System.out.println(clz.getName()); //输出java.lang.String
        }
    }
}
```

动态代理

熟悉设计模式的人对于代理模式可 能都不陌生。代理对象和被代理对象一般实现相同的接口，调用者与代理对象进行交互。代理的存在对于调用者来说是透明的，调用者看到的只是接口。代理对象则可以封装一些内部的处理逻辑，如访问控制、远程通信、日志、缓存等。比如一个对象访问代理就可以在普通的访问机制之上添加缓存的支持。这种模式在 RMI和 EJB中 都得到了广泛的使用。传统的代理模式的实现，需要在源代码中添加一些附加的类。这些类一般是手写或是通过工具来自动生成。JDK 5 引入的动态代理机制，允许开发人员在运行时刻动态的创建出代理类及

其对象。在运行时刻，可以动态创建出一个实现了多个接口的代理类。每个代理类的对象都会关联一个表示内部处理逻辑的 InvocationHandler接口的实现。当使用者调用了代理对象所代理的接口中的方法的时候，这个调用的信息会被传递给InvocationHandler的invoke方法。在 invoke方法的参数中可以获取到代理对象、方法对应的Method对象和调用的实际参数。invoke方法的返回值被返回给使用者。这种做法实际上相当于对方法调用进行了拦截。熟悉AOP的人对这种使用模式应该不陌生。但是这种方式不需要依赖 AspectJ等AOP框架。

下面的代码用来代理一个实现了 List 接口的对象。所实现的功能也非常简单，那就是禁止使用 List 接口中的 add 方法。如果在 getList 中传入一个实现 List 接口的对象，那么返回的实际就是一个代理对象 ,尝试在该对象上调用 add 方法就会抛出来异常。

```
public List getList(final List list) {
   return                                                    (List)
Proxy.newProxyInstance(DummyProxy.class.getClassLoader(), new Class[]
{ List.class },
       new InvocationHandler() {
          public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
              if ("add".equals(method.getName())) {
                 throw new UnsupportedOperationException();
              }
              else {
                 return method.invoke(list, args);
              }
          }
       });
 }
```

这里的实际流程是，当代理对象的 add 方法被调用的时候，InvocationHandler 中的 invoke 方法会被调用。参数 method 就包含了调用的基本信息。因为方法名称是 add，所以会抛出相关的异常。如果调用的是其它方法的话，则执行原来的逻辑。

使用案例

Java 反射 API 的存在，为 Java 语言添加了一定程度上的动态性，可以实现某些动态语言中的功能。比如在 JavaScript 的代码中，可以通过 obj["set" + propName]()来根据变量 propName 的值找到对应的方法进行调用。虽然在 Java 源代码中不能这么写，但是通过反射 API 同样可以实现类似的功能。这对于处理某些遗留代码来说是有帮助的。比如所需要使用的类有多个版本，每个版本所提供的方法名称和参数不尽相同。而调用代码又必须与这些不同的版本都能协同工作，就可以通过反射 API 来依

次检查实际的类中是否包含某个方法来选择性的调用。

Java 反射 API 实际上定义了一种相对于编译时刻而言更加松散的契约。如果被调用的 Java 对象中并不包含某个方法，而在调用者代码中进行引用的话，在编译时刻就会出现错误。而反射 API 则可以把这样的检查推迟到运行时刻来完成。通过把 Java 中的字节代码增强、类加载器和反射 API 结合起来，可以处理一些对灵 活性要求很高的场景。

在有些情况下，可能会需要从远端加载一个 Java 类来执行。比如一个客户端 Java 程序可以通过网络从服务器端下载 Java 类来执行，从而可以实现自动更新的机制。当代码逻辑需要更新的时候，只需要部署一个新的 Java 类到服务器端即可。一般的做法是通过自定义类加载器下载了类字节代码之后，定义出 Class 类的对象，再通过 newInstance 方法就可以创建出实例了。不过这种做法要求客户端和服务器端都具有某个接口的定义，从服务器端下载的是这个接口的实现。这样的话才能在客户端进行所需的类型转换，并通过接口来使用这个对象实例。如果希望客户端和服务器端采用更加松散的契约的话，使用反射 API 就可以了。两者之间的契约只需要在方法的名称和参数这个级别就足够了。服务器端 Java 类并不需要实现特定的接口，可以是一般的 Java 类。

动态代理的使用场景就更加广泛了。需要使用AOP中的方法拦截功能的地方都可以用到动态代理。Spring框架的 AOP实现 默认也使用动态代理。不过JDK中的动态代理只支持对接口的代理，不能对一个普通的Java类提供代理。不过这种实现在大部分的时候已经够用了。

参考资料

- Classworking toolkit: Reflecting generics
- Decorating with dynamic proxies

8

Java I/O

在应用程序中，通常会涉及到两种类型的计算：CPU 计算和 I/O 计算。对于大多数应用来说，花费在等待 I/O 上的时间是占较大比重的。通常需要等待速度较慢的磁盘或是网络连接完成 I/O 请求，才能继续后面的 CPU 计算任务。因此提高 I/O 操作的效率对应用的性能有较大的帮助。本文将介绍 Java 语言中与 I /O 操作相关的内容，包括基本的 Java I/O 和 Java NIO，着重于基本概念和最佳实践。

流

Java语言提供了多个层次不同的概念来对I/O操作进行抽象。Java I/O中最早的概念是流，包括输入流和输出流，早在JDK 1.0 中就存在了。简单的来说，流是一个连续的字节的序列。输入流是用来读取这个序列，而输出流则构建这个序列。InputStream 和 OutputStream所操纵的基本单元就是字节。每次读取和写入单个字节或是字节数组。如果从字节的层次来处理数据类型的话，操作会非常繁琐。可以用更易使用的流实现来包装基本的字节流。如果想读取或输出Java的基本数据类型，可以使用 DataInputStream和 DataOutputStream。它们所提供的类似readFloat和writeDouble这样的方法，会让处理基本数据类型变得很简单。如果希望读取或写入的是Java中的对象的话，可以使用 ObjectInputStream和 ObjectOutputStream。它们与对象的 序列化 机制一起，可以实现Java对象状态的持久化和数据传递。基本流所提供的对于输入和输出的控制比较弱。InputStream只提供了顺序读取、跳过部分字节和标记/重置的支持，而OutputStream则只能顺序输出。

流的使用

由于I/O操作所对应的实体在系统中都是有限的资源，需要妥善的进行管理。每个打开的流都需要被正确的关闭以释放资源。所遵循的原则是谁打开谁释放。如果一个流只在某个方法体内使用，则通过finally语句或是JDK 7 中的 try-with-resources语句来确保在方法返回之前，流被正确的关闭。如果一个方法只是作为流的使用者，就不需要考虑流的关闭问题。典型的情况是在servlet实现中并不需要关闭

HttpServletResponse中的输出流。如果你的代码需要负责打开一个流，并且需要在不同的对象之间进行传递的话，可以考虑使用 Execute Around Method模式。如下面的代码所示：

```
public void use(StreamUser user) {
    InputStream input = null;
    try {
        input = open();
        user.use(input);
    } catch(IOException e) {
        user.onError(e);
    } finally {
        if (input != null) {
            try {
                input.close();
            } catch (IOException e) {
                user.onError(e);
            }
        }
    }
 }
```

如上述代码中所看到的一样，由专门的类负责流的打开和关闭。流的使用者StreamUser 并不需要关心资源释放的细节，只需要对流进行操作即可。

在使用输入流的过程中，经常会遇到需要复用一个输入流的情况，即多次读取一个输入流中的内容。比如通过URL.openConnection方法打开了一 个远端站点连接的输入流，希望对其中的内容进行多次处理。这就需要把一个InputStream对象在多个对象中传递。为了保证每个使用流的对象都能获取到正确的内容，需要对流进行一定的处理。通常有两种解决的办法，一种是利用InputStream的标记支持。如果一个流支持标记的话（通过 markSupported方法判断），就可以在流开始的地方通过 mark方法添加一个标记，当完成一次对流的使用之后，通过 reset方法就可以把流的读取位置重置到上次标记的位置，即流开始的地方。如此反复，就可以复用这个输入流。大部分输入流的实现是不支持标记的。可以通过 BufferedInputStream进行包装来支持标记。

```
private InputStream prepareStream(InputStream ins) {
    BufferedInputStream buffered = new BufferedInputStream(ins);
    buffered.mark(Integer.MAX_VALUE);
    return buffered;
}
private void resetStream(InputStream ins) throws IOException {
    ins.reset();
```

```
    ins.mark(Integer.MAX_VALUE);
}
```

如上面的代码所示，通过 prepareStream 方法可以用一个 BufferedInputStream 来包装基本的 InputStream。通过 mark 方法在流开始的时候添加一个标记，允许读入 Integer.MAX_VALUE 个字节。每次流使用完成之后，通过 resetStream 方法重置即可。

另外一种做法是把输入流的内容转换成字节数组，进而转换成输入流的另外一个实现 ByteArrayInputStream。这样做的好处是使用字节数组作为参数传递的格式要比输入流简单很多，可以不需要考虑资源相关的问题。另外也可以尽早的关闭原始的输入流，而无需等待所有使用流的操作完成。这两种做法的思路其实是相似的。BufferedInputStream在内部也创建了一个字节数组来保存从原始输入流中读入的内容。

```
private byte[] saveStream(InputStream input) throws IOException {
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    ReadableByteChannel readChannel = Channels.newChannel(input);
    ByteArrayOutputStream output = new ByteArrayOutputStream(32 * 1024);
    WritableByteChannel writeChannel = Channels.newChannel(output);
    while ((readChannel.read(buffer)) > 0 || buffer.position() != 0) {
        buffer.flip();
        writeChannel.write(buffer);
        buffer.compact();
    }
    return output.toByteArray();
}
```

上面的代码中 saveStream 方法把一个 InputStream 保存为字节数组。

缓冲区

由于流背后的数据有可能比较大，在实际的操作中，通常会使用缓冲区来提高性能。传统的缓冲区的实现是使用数组来完成。比如经典的从InputStream到OutputStream的复制的 实现 ，就是使用一个字节数组作为中间的缓冲区。NIO中引入的 Buffer类及其子类，可以很方便的用来创建各种基本数据类型的缓冲区。相对于数组而言，Buffer类及其子类提供了更加丰富的方法来对其中的数据进行操作。后面会提到的通道也使用Buffer类进行数据传递。

在Buffer上进行的元素添加和删除操作，都围绕 3 个属性 position、limit和 capacity展开，分别表示Buffer当前的读写位置、可用的读写范围和容量限制。容量限制是在创建的时候指定的。 Buffer提供的get/put方法都有相对和绝对两种形式。相对读写时

的位置是相对于position的值，而绝对读写则需要指定起始的序号。在使用Buffer的常见错误就是在读写操作时没有考虑到这 3 个元素的值，因为大多数时候都是使用的是相对读写操作 ,而position的值可能早就发生了变化。一些应该注意的地方包括：将数据读入缓冲区之前， 需要调用 clear方法；将缓冲区中的数据输出之前，需要调用 flip方法。

```
ByteBuffer buffer = ByteBuffer.allocate(32);
CharBuffer charBuffer = buffer.asCharBuffer();
String           content           =           charBuffer.put("Hello
").put("World").flip().toString();
System.out.println(content);
```

上面的代码展示了 Buffer 子类的使用。首先可以在已有的 ByteBuffer 上面创建出其它数据类型的缓冲区视图，其次 Buffer 子类的很多方法是可以级联的，最后是要注意 flip 方法的使用。

字符与编码

在程序中，总是免不了与字符打交道，毕竟字符是用户直接可见的信息。而与字符处理直接相关的就是编码。相信不少人都曾经为了程序中的乱码问题而困扰。要弄清楚这个问题，就需要理解字符集和编码的概念。字符集，顾名思义，就是字符的集合。一个字符集中所包含的字符通常与地区和语言有关。字符集中的每个字符通常会有一个整数编码与其对应。常见的字符集有 ASCII、ISO-8859-1 和 Unicode 等。对于字符集中的每个字符，为了在计算机中表示，都需要转换某种字节的序列，即该字符的编码。同一个字符集可以有不同的编码方式。如果某种编码格式产生的字节序列，用另外一种编码格式来解码的话，就可能会得到错误的字符，从而产生乱码的情况。所以将一个字节序列转换成字符串的时候，需要知道正确的编码格式。

NIO中的 java.nio.charset包提供了与字符集相关的类，可以用来进行编码和解码。其中的 CharsetEncoder和 CharsetDecoder允许对编码和解码过程进行精细的控制 ,如处理非法的输入以及字符集中无法识别的字符等。通过这两个类可以实现字符内容的过滤。比如应用程序在设计的时候就只支持某种字符集，如果用户输入了其它字符集中的内容，在界面显示的时候就是乱码。对于这种情况，可以在解码的时候忽略掉无法识别的内容。

```
tring input = "你123好";
Charset charset = Charset.forName("ISO-8859-1");
CharsetEncoder encoder = charset.newEncoder();
encoder.onUnmappableCharacter(CodingErrorAction.IGNORE);
```

```
CharsetDecoder decoder = charset.newDecoder();
CharBuffer buffer = CharBuffer.allocate(32);
buffer.put(input);
buffer.flip();
try {
    ByteBuffer byteBuffer = encoder.encode(buffer);
    CharBuffer cbuf = decoder.decode(byteBuffer);
    System.out.println(cbuf);  //输出123
} catch (CharacterCodingException e) {
    e.printStackTrace();
}
```

上面的代码中，通过使用 ISO-8859-1 字符集的编码和解码器，就可以过滤掉字符串中不在此字符集中的字符。

Java I/O在处理字节流字之外，还提供了处理字符流的类，即 Reader/Writer类及其子类，它们所操纵的基本单位是char类型。在字节和字符之间的桥梁就是编码格式。通过编码器来完成这两者之间的转换。在创建 Reader/Writer子类实例的时候，总是应该使用两个参数的构造方法，即显式指定使用的字符集或编码解码器。如果不显式指定，使用的是JVM的默认字符集，有可能在其它平台上产生错误。

通道

通道作为 NIO 中的核心概念，在设计上比之前的流要好不少。通道相关的很多实现都是接口而不是抽象类。通道本身的抽象层次也更加合理。通道表示的是对支持 I/O 操作的实体的一个连接。一旦通道被打开之后，就可以执行读取和写入操作，而不需要像流那样由输入流或输出流来分别进行处理。与流相比，通道的操作使用的是 Buffer 而不是数组，使用更加方便灵活。通道的引入提升了 I/O 操作的灵活性和性能，主要体现在文件操作和网络操作上。

文件通道

对文件操作方面 ,文件通道 FileChannel提供了与其它通道之间高效传输数据的能力，比传统的基于流和字节数组作为缓冲区的做法，要来得简单和快速。比如下面的把一个网页的内容保存到本地文件的实现。

```
FileOutputStream output = new FileOutputStream("baidu.txt");
FileChannel channel = output.getChannel();
URL url = new URL("http://www.baidu.com");
InputStream input = url.openStream();
ReadableByteChannel readChannel = Channels.newChannel(input);
```

```
channel.transferFrom(readChannel, 0, Integer.MAX_VALUE);
```

文件通道的另外一个功能是对文件的部分片段进行加锁。当在一个文件上的某个片段加上了排它锁之后，其它进程必须等待这个锁释放之后，才能访问该文件的这个片段。文件通道上的锁是由 JVM 所持有的 ,因此适合于与其它应用程序协同时使用。比如当多个应用程序共享某个配置文件的时候，如果 Java 程序需要更新此文件，则可以首先获取该文件上的一个排它锁，接着进行更新操作，再释放锁即可。这样可以保证文件更新过程中不会受到其它程序的影响。

另外一个在性能方面有很大提升的功能是 内存映射文件 的支持。通过FileChannel的 map方法可以创建出一个 MappedByteBuffer对象，对这个缓冲区的操作都会直接反映到文件内容上。这点尤其适合对大文件进行读写操作。

套接字通道

在套接字通道方面的改进是提供了对非阻塞I/O和多路复用I/O的支持。传统的流的I/O操作是阻塞式的。在进行I/O操作的时候，线程会处于阻塞状态等待操作完成。NIO 中引入了非阻塞 I/O 的支持，不过只限于套接字 I/O 操作。所有继承自 SelectableChannel的通道类都可以通过 configureBlocking方法来设置是否采用非阻塞模式。在非阻塞模式下，程序可以在适当的时候查询是否有数据可供读取。一般是通过定期的轮询来实现的。

多路复用I/O是一种新的I/O编程模型。传统的套接字服务器的处理方式是对于每一个客户端套接字连接，都新创建一个线程来进行处理。创建线程是很耗时的操作，而有的实现会采用线程池。不过一个请求一个线程的处理模型并不是很理想。原因在于耗费时间创建的线程，在大部分时间可能处于等待的状态。而多路复用 I/O的基本做法是由一个线程来管理多个套接字连接。该线程会负责根据连接的状态，来进行相应的处理。多路复用I/O依靠操作系统提供的 select或相似系统调用的支持，选择那些已经就绪的套接字连接来处理。可以把多个非阻塞I/O通道注册在某个 Selector上，并声明所感兴趣的操作类型。每次调用Selector的 select方法，就可以选择到某些感兴趣的操作已经就绪的通道的集合，从而可以进行相应的处理。如果要执行的处理比较复杂，可以把处理转发给其它的线程来执行。

下面是一个简单的使用多路复用 I/O 的服务器实现。当有客户端连接上的时候，服务器会返回一个 Hello World 作为响应。

```
private static class IOWorker implements Runnable {
    public void run() {
        try {
            Selector selector = Selector.open();
            ServerSocketChannel channel = ServerSocketChannel.open();
            channel.configureBlocking(false);
            ServerSocket socket = channel.socket();
            socket.bind(new InetSocketAddress("localhost", 10800));
            channel.register(selector, channel.validOps());
            while (true) {
                selector.select();
                Iterator iterator = selector.selectedKeys().iterator();
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    iterator.remove();
                    if (!key.isValid()) {
                        continue;
                    }
                    if (key.isAcceptable()) {
                        ServerSocketChannel ssc = (ServerSocketChannel)
key.channel();
                        SocketChannel sc = ssc.accept();
                        sc.configureBlocking(false);
                        sc.register(selector, sc.validOps());
                    }
                    if (key.isWritable()) {
                        SocketChannel client = (SocketChannel)
key.channel();
                        Charset charset = Charset.forName("UTF-8");
                        CharsetEncoder encoder = charset.newEncoder();
                        CharBuffer charBuffer = CharBuffer.allocate(32);
                        charBuffer.put("Hello World");
                        charBuffer.flip();
                        ByteBuffer content = encoder.encode(charBuffer);
                        client.write(content);
                        key.cancel();
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上面的代码给出的只是非常简单的示例程序，只是展示了多路复用I/O的基本使用方式。在开发复杂网络应用程序的时候，使用一些Java NIO网络应用框架会让你事半功

倍。目前来说最流行的两个框架是 Apache MINA和 Netty。在使用了Netty之后 ,Twitter 的 搜索功能速度提升达到了 3 倍之多 。网络应用开发人员都可以使用这两个开源的优秀框架。

参考资料

- Java 6 I/O-related APIs & Developer Guides
- Top Ten New Things You Can Do with NIO
- Building Highly Scalable Servers with Java NIO

9

Java 安全

安全性是Java应用程序的非功能性需求的重要组成部分，如同其它的非功能性需求一样，安全性很容易被开发人员所忽略。当然，对于Java EE的开发人员来说，安全性的话题可能没那么陌生，用户认证和授权可能是绝大部分Web应用都有的功能。类似 Spring Security这样的框架，也使得开发变得更加简单。本文并不会讨论Web应用的安全性，而是介绍Java安全一些底层和基本的内容。

认证

用户认证是应用安全性的重要组成部分，其目的是确保应用的使用者具有合法的身份。 Java安全中使用术语主体（Subject）来表示访问请求的来源。一个主体可以是任何的实体。一个主体可以有多个不同的身份标识（Principal）。比如一个应用的用户这类主体，就可以有用户名、身份证号码和手机号码等多种身份标识。除了身份标识之外，一个主体还可以有公开或是私有的安全相关的凭证（Credential），包括密码和密钥等。

典型的用户认证过程是通过登录操作来完成的。在登录成功之后，一个主体中就具备了相应的身份标识。Java提供了一个可扩展的登录框架，使得应用开发人员可以很容易的定制和扩展与登录相关的逻辑。登录的过程由 LoginContext启动。在创建LoginContext的时候需要指定一个登录配置（Configuration）的名称。该登录配置中包含了登录所需的多个 LoginModule的信息。每个LoginModule实现了一种登录方式。当调用LoginContext的 login方法的时候，所配置的每个LoginModule会被调用来执行登录操作。如果整个登录过程成功，则通过 getSubject方法就可以获取到包含了身份标识信息的主体。开发人员可以实现自己的LoginModule来定制不同的登录逻辑。

每个LoginModule的登录方式由两个阶段组成。第一个阶段是在 login方法的实现中。这个阶段用来进行必要的身份认证，可能需要获取用户的输入，以及通过数据库、网络操作或其它方式来完成认证。当认证成功之后，把必要的信息保存起来。如果认证失败，则抛出相关的异常。第二阶段是在 commit或 abort方 法中。由于一个登

录过程可能涉及到多个LoginModule。LoginContext会根据每个LoginModule的认证结果以及相关的配置信息 来确定本次登录是否成功。LoginContext用来判断的依据是每个LoginModule对整个登录过程的必要性，分成必需、必要、充分和可选这四种情况。如果登录成功，则每个LoginModule的commit方法会被调用，用来把身份标识关联到主体上。如果登录失败，则LoginModule 的abort方法会被调用，用来清除之前保存的认证相关信息。

在LoginModule进行认证的过程中，如果需要获取用户的输入，可以通过 CallbackHandler和对应的Callback来完成。每个 Callback可以用来进行必要的数据传递。典型的启动登录的过程如下：

```
public Subject login() throws LoginException {
    TextInputCallbackHandler callbackHandler = new
TextInputCallbackHandler();
    LoginContext lc = new LoginContext("SmsApp", callbackHandler);
    lc.login();
    return lc.getSubject();
}
```

这里的 SmsApp 是登录配置的名称，可以在配置文件中找到。该配置文件的内容也很简单。

```
SmsApp {
    security.login.SmsLoginModule required;
};
```

这里声明了使用 security.login.SmsLoginModule 这个登录模块 ,而且该模块是必需的。配置文件可以通过启动程序时的参数 java.security.auth.login.config 来指定，或修改 JVM 的默认设置。下面看看 SmsLoginModule 的核心方法 login 和 commit。

```
public boolean login() throws LoginException {
    TextInputCallback phoneInputCallback = new TextInputCallback("Phone
number: ");
    TextInputCallback smsInputCallback = new TextInputCallback("Code:
");
    try {
        handler.handle(new Callback[] {phoneInputCallback,
smsInputCallback});
    } catch (Exception e) {
        throw new LoginException(e.getMessage());
    }
    String code = smsInputCallback.getText();
    boolean isValid = code.length() > 3; //此处只是简单的进行验证。
    if (isValid) {
        phoneNumber = phoneInputCallback.getText();
```

```
    }
    return isValid;
}
public boolean commit() throws LoginException {
    if (phoneNumber != null) {
    subject.getPrincipals().add(new PhonePrincipal(phoneNumber));
    return true;
}
    return false;
}
```

这里使用了两个 TextInputCallback来获取用户的输入。当用户输入的编码有效的时候，就把相关的信息记录下来，此处是用户的手机号码。在commit方法中，就把该手机号码作为用户的身份标识与主体关联起来。

权限控制

在验证了访问请求来源的合法身份之后，另一项工作是验证其是否具有相应的权限。权限由 Permission及其子类来表示。每个权限都有一个名称，该名称的含义与权限类型相关。某些权限有与之对应的动作列表。比较典型的是文件操作权限 FilePermission，它的名称是文件的路径，而它的动作列表则包括读取、写入和执行等。Permission类中最重要的是 implies方法，它定义了权限之间的包含关系，是进行验证的基础。

权限控制包括管理和验证两个部分。管理指的是定义应用中的权限控制策略，而验证指的则是在运行时刻根据策略来判断某次请求是否合法。策略可以与主体关联，也可以没有关联。策略由 Policy来表示，JDK提供了基于文件存储的基本实现。开发人员也可以提供自己的实现。在应用运行过程中，只可能有一个Policy处于生效的状态。验证部分的具体执行者是 AccessController，其中的 checkPermission方法用来验证给定的权限是否被允许。在应用中执行相关的访问请求之前，都需要调用checkPermission 方法来进行验证。如果验证失败的话，该方法会抛出 AccessControlException异常。JVM中内置提供了一些对访问关键部分内容的访问控制检查，不过只有在启动应用的时通过参数-Djava.security.manager启用了安全管理器之后才能生效，并与策略相配合。

与访问控制相关的另外一个概念是特权动作。特权动作只关心动作本身所要求的权限是否具备，而并不关心调用者是谁。比如一个写入文件的特权动作，它只要求对该文件有写入权限即可，并不关心是谁要求它执行这样的动作。特权动作根据是否

抛出受检异常，分为 PrivilegedAction和 PrivilegedExceptionAction。这两个接口都只有一个run方法用来执行相关的动作，也可以向调用者返回结果。通过AccessController的doPrivileged方法就可以执行特权动作。

Java安全使用了保护域的概念。每个保护域都包含一组类、身份标识和权限，其意义是在当访问请求的来源是这些身份标识的时候，这些类的实例就自动具有给定的这些权限。保护域的权限既可以是固定，也可以根据策略来动态变化。ProtectionDomain类用来表示保护域，它的两个构造方法分别用来支持静态和动态的权限。一般来说，应用程序通常会涉及到系统保护域和应用保护域。不少的方法调用可能会跨越多个保护域的边界。因此，在AccessController进行访问控制验证的时候，需要考虑当前操作的调用上下文，主要指的是方法调用栈上不同方法所属于 的不同保护域。这个调用上下文一般是与当前线程绑定在一起的。通过AccessController 的 getContext 方法可以获取到表示调用上下文的 AccessControlContext对象，相当于访问控制验证所需的调用栈的一个快照。在有些情况下，会需要传递此对象以方便在其它线程中进行访问控制验证。

考虑下面的权限验证代码：

```
Subject subject = new Subject();
ViewerPrincipal principal = new ViewerPrincipal("Alex");
subject.getPrincipals().add(principal);
Subject.doAsPrivileged(subject, new PrivilegedAction<Object>() {
    public Object run() {
        new Viewer().view();
        return null;
    }
}, null);
```

这里创建了一个新的 Subject 对象并关联上身份标识。通常来说，这个过程是由登录操作来完成的。通过 Subject 的 doAsPrivileged 方法就可以执行一个特权动作。Viewer 对象的 view 方法会使用 AccessController 来检查是否具有相应的权限。策略配置文件的内容也比较简单，在启动程序的时候通过参数 java.security.auth.policy 指定文件路径即可。

```
grant Principal security.access.ViewerPrincipal "Alex" {
    permission security.access.ViewPermission "CONFIDENTIAL";
};//这里把名称为CONFIDENTIAL的ViewPermission授权给了身份标识为Alex的主体。
```

加密、解密与签名

构建安全的Java应用离不开加密和解密。Java的密码框架采用了常见的服务提供者架构，以提供所需的可扩展性和互操作性。该密码框架提供了一系列常用的服务，包括加密、数字签名和报文摘要等。这些服务都有服务提供者接口（SPI），服务的实现者只需要实现这些接口，并注册到密码框架中即可。比如加密服务 Cipher的SPI接口就是 CipherSpi。每个服务都可以有不同的算法来实现。密码框架也提供了相应的工厂方法用来获取到服务的实例。比如想使用采用MD5 算法的报文摘要服务，只需要调用MessageDigest.getInstance("MD5")即可。

加密和解密过程中并不可少的就是密钥（Key）。加密算法一般分成对称和非对称两种。对称加密算法 使用同一个密钥进行加密和解密；而 非对称加密算法 使用一对公钥和私钥，一个加密的时候，另外一个就用来解密。不同的加密算法，有不同的密钥。对称加密算法使用的是 SecretKey，而非对称加密算法则使用 PublicKey 和 PrivateKey。与密钥Key对应的另一个接口是 KeySpec，用来描述不同算法的密钥的具体内容。比如一个典型的使用对称加密的方式如下：

```
KeyGenerator generator = KeyGenerator.getInstance("DES");
SecretKey key = generator.generateKey();
saveFile("key.data", key.getEncoded());
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, key);
String text = "Hello World";
byte[] encrypted = cipher.doFinal(text.getBytes());
saveFile("encrypted.bin", encrypted);
```

加密的时候首先要生成一个密钥，再由 Cipher 服务来完成。可以把密钥的内容保存起来，方便传递给需要解密的程序。

```
byte[] keyData = getData("key.data");
SecretKeySpec keySpec = new SecretKeySpec(keyData, "DES");
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.DECRYPT_MODE, keySpec);
byte[] data = getData("encrypted.bin");
byte[] result = cipher.doFinal(data);
```

解密的时候先从保存的文件中得到密钥编码之后的内容，再通过 SecretKeySpec获取到密钥本身的内容，再进行解密。

报文摘要 的目的在于防止信息被有意或无意的修改。通过对原始数据应用某些算法，可以得到一个校验码。当收到数据之后，只需要应用同样的算法，再比较校验

码是否一致，就可以判断数据是否被修改过。相对原始数据来说，校验码长度更小，更容易进行比较。消息认证码（Message Authentication Code）与报文摘要类似，不同的是计算的过程中加入了密钥，只有掌握了密钥的接收者才能验证数据的完整性。

使用公钥和私钥就可以实现数字签名的功能。某个发送者使用私钥对消息进行加密，接收者使用公钥进行解密。由于私钥只有发送者知道，当接收者使用公钥解密成功之后，就可以判定消息的来源肯定是特定的发送者。这就相当于发送者对消息进行了签名。数字签名由 Signature服务提供，签名和验证的过程都比较直接。

```
Signature signature = Signature.getInstance("SHA1withDSA");
KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("DSA");
KeyPair keyPair = keyGenerator.generateKeyPair();
PrivateKey privateKey = keyPair.getPrivate();
signature.initSign(privateKey);
byte[] data = "Hello World".getBytes();
signature.update(data);
byte[] signatureData = signature.sign(); //得到签名
PublicKey publicKey = keyPair.getPublic();
signature.initVerify(publicKey);
signature.update(data);
boolean result = signature.verify(signatureData); //进行验证
```

验证数字签名使用的公钥可以通过文件或证书的方式来进行发布。

安全套接字连接

在各种数据传输方式中，网络传输目前使用较广，但是安全隐患也更多。安全套接字连接指的是对套接字连接进行加密。加密的时候可以选择对称加密算法。但是如何在发送者和接收者之间安全的共享密钥，是个很麻烦的问题。如果再用加密算法来加密密钥，则成为了一个循环问题。非对称加密算法则适合于这种情况。私钥自己保管，公钥则公开出去。发送数据的时候，用私钥加密，接收者用公开的公钥解密；接收数据的时候，则正好相反。这种做法解决了共享密钥的问题，但是另外的一个问题是如何确保接收者所得到的公钥确实来自所声明的发送者，而不是伪造的。为此，又引入了证书的概念。证书中包含了身份标识和对应的公钥。证书由用户所信任的机构签发，并用该机构的私钥来加密。在有些情况下，某个证书签发机构的真实性会需要由另外一个机构的证书来证明。通过这种证明关系，会形成一个证书的链条。而链条的根则是公认的值得信任的机构。只有当证书链条上的所有证书都被信任的时候，才能信任证书中所给出的公钥。

日常开发中比较常接触的就是 HTTPS，即安全的HTTP连接。大部分用Java程序访问采用HTTPS网站时出现的错误都与证书链条相关。有些网站采用的不是由正规安全机构签发的证书，或是证书已经过期。如果必须访问这样的HTTPS网站的话，可以提供自己的套接字工厂和主机名验证类来绕过去。另外一种做法是通过 keytool工具把证书导入到系统的信任证书库之中。

```
URL url = new URL("https://localhost:8443");
SSLContext context = SSLContext.getInstance("TLS");
context.init(new KeyManager[] {}, new TrustManager[] {new
MyTrustManager()}, new SecureRandom());HttpsURLConnection connection =
(HttpsURLConnection) url.openConnection();
connection.setSSLSocketFactory(context.getSocketFactory());
connection.setHostnameVerifier(new MyHostnameVerifier());
```

这里的MyTrustManager实现了 X509TrustManager接口 ,但是所有方法都是默认实现。而MyHostnameVerifier实现了 HostnameVerifier接口 ,其中的verify方法总是返回true。

参考资料

- Java安全体系结构 、Java密码框架(JCA)参考指南 、Java认证和授权服务(JAAS)参考指南 、Java安全套接字扩展 (JSSE) 参考指南

10

Java 对象序列化与 RMI

对于一个存在于Java虚拟机中的对象来说，其内部的状态只保持在内存中。JVM停止之后，这些状态就丢失了。在很多情况下，对象的内部状态是需要被持久化下来的。提到持久化，最直接的做法是保存到文件系统或是数据库之中。这种做法一般涉及到自定义存储格式以及繁琐的数据转换。对象关系映射（Object-relational mapping）是一种典型的用关系数据库来持久化对象的方式，也存在很多直接存储对象的对象数据库。对象序列化机制（object serialization）是Java语言内建的一种对象持久化方式，可以很容易的在JVM中的活动对象和字节数组（流）之间进行转换。除了可以很简单 的实现持久化之外，序列化机制的另外一个重要用途是在远程方法调用中，用来对开发人员屏蔽底层实现细节。

基本的对象序列化

由于Java提供了良好的默认支持，实现基本的对象序列化是件比较简单的事。待序列化的Java类只需要实现 Serializable接口即可。Serializable仅是一个标记接口，并不包含任何需要实现的具体方法。实现该接口只是为了声明该Java类的对象是可以被序列化的。实际的序列化和反序列化工作是通过 ObjectOuputStream和 ObjectInputStream来完成的。ObjectOutputStream的 writeObject方法可以把一个Java对象写入到流中，ObjectInputStream的 readObject方 法可以从流中读取一个Java对象。在写入和读取的时候，虽然用的参数或返回值是单个对象，但实际上操纵的是一个对象图，包括该对象所引用的其它对象 以 及这些对象所引用的另外的对象。Java会自动帮你遍历对象图并逐个序列化。除了对象之外，Java中的基本类型和数组也是可以通过 ObjectOutputStream和ObjectInputStream来序列化的。

```
try {
    User user = new User("Alex", "Cheng");
    ObjectOutputStream    output    =    new    ObjectOutputStream(new
FileOutputStream("user.bin"));
    output.writeObject(user);
    output.close();
} catch (IOException e) {
```

```
    e.printStackTrace();
}

try {
    ObjectInputStream input = new ObjectInputStream(new
                    FileInputStream("user.bin"));
    User user = (User) input.readObject();
    System.out.println(user);
} catch (Exception e) {
    e.printStackTrace();
}
```

上面的代码给出了典型的把 Java 对象序列化之后保存到磁盘上，以及从磁盘上读取的基本方式。User 类只是声明了实现 Serializable 接口。

在默认的序列化实现中，Java对象中的非静态和非瞬时域都会被包括进来，而与域的可见性声明没有关系。这可能会导致某些不应该出现的域被包含在序列化之后的字节数组中，比如密码等隐私信息。由于Java对象序列化之后的格式是固定的，其它人可以很容易的从中分析出其中的各种信息。对于这种情况，一种解决办法是把域声明为瞬时的，即使用 transient 关键词。另外一种做法是添加一个 serialPersistentFields? 域来声明序列化时要包含的域。从这里可以看到在Java序列化机制中的这种仅在书面层次上定义的契约。声明序列化的域必须使用固定的名称和类型。在后面还可以看到其它类似这样的契约。虽然Serializable只是一个标记接口，但它其实是包含有不少隐含的要求。下面的代码给出了 serialPersistentFields的声明示例，即只有firstName这个域是要被序列化的。

```
private static final ObjectStreamField[] serialPersistentFields = {
    new ObjectStreamField("firstName", String.class)
};
```

自定义对象序列化

基本的对象序列化机制让开发人员可以在包含哪些域上进行定制。如果想对序列化的过程进行更加细粒度的控制，就需要在类中添加 writeObject 和对应的 readObject 方法。这两个方法属于前面提到的序列化机制的隐含契约的一部分。在通过 ObjectOutputStream 的 writeObject 方法写入对象的时候，如果这个对象的类中定义了 writeObject 方法，就会调用该方法，并把当前 ObjectOutputStream 对象作为参数传递进去。writeObject 方法中一般会包含自定义的序列化逻辑，比如在写入之前修改域的值，或是写入额外的数据等。对于 writeObject 中添加的逻辑，在对应的 readObject 中都需要反转过来，与之对应。

在添加自己的逻辑之前，推荐的做法是先调用Java的默认实现。在writeObject方法中通过ObjectOutputStream的 defaultWriteObject来完成，在readObject方法则通过ObjectInputStream的 defaultReadObject来实现。下面的代码在对象的序列化流中写入了一个额外的字符串。

```
private void writeObject(ObjectOutputStream output) throws IOException
{
    output.defaultWriteObject();
    output.writeUTF("Hello World");
}
private void readObject(ObjectInputStream input) throws IOException,
ClassNotFoundException {
    input.defaultReadObject();
    String value = input.readUTF();
    System.out.println(value);
}
```

序列化时的对象替换

在有些情况下，可能会希望在序列化的时候使用另外一个对象来代替当前对象。其中的动机可能是当前对象中包含了一些不希望被序列化的域，比如这些域都是从另外一个域派生而来的；也可能是希望隐藏实际的类层次结构；还有可能是添加自定义的对象管理逻辑，如保证某个类在JVM中只有一个实例。相对于把无关的域都设成transient来说，使用对象替换是一个更好的选择，提供了更多的灵活性。替换对象的作用类似于Java EE中会使用到的 传输对象（Transfer Object）。

考虑下面的例子，一个订单系统中需要把订单的相关信息序列化之后，通过网络来传输。订单类 Order 引用了客户类 Customer。在默认序列化的情况下，Order 类对象被序列化的时候，其引用的 Customer 类对象也会被序列化，这可能会造成用户信息的泄露。对于这种情况，可以创建一个另外的对象来在序列化的时候替换当前的 Order 类的对象，并把用户信息隐藏起来。

```
private static class OrderReplace implements Serializable {
    private static final long serialVersionUID = 4654546423735192613L;
    private String orderId;
    public OrderReplace(Order order) {
        this.orderId = order.getId();
    }
    private Object readResolve() throws ObjectStreamException {
        //根据orderId查找Order对象并返回
    }
}
```

这个替换对象类 OrderReplace 只保存了 Order 的 ID。在 Order 类的 writeReplace 方法中返回了一个 OrderReplace 对象。这个对象会被作为替代写入到流中。同样的，需要在 OrderReplace 类中定义一个 readResolve 方法，用来在读取的时候再转换回 Order 类对象。这样对调用者来说，替换对象的存在就是透明的。

```
private Object writeReplace() throws ObjectStreamException {
    return new OrderReplace(this);
}
```

序列化与对象创建

在通过 ObjectInputStream 的 readObject 方法读取到一个对象之后，这个对象是一个新的实例，但是其构造方法是没有被调用的，其中的域的初始化代码也没有被执行。对于那些没有被序列化的域，在新创建出来的对象中的值都是默认的。也就是说，这个对象从某种角度上来说是不完备的。这有可能会造成一些隐含的错误。调用者并不知道对象是通过一般的 new 操作符来创建的，还是通过反序列化所得到的。解决的办法就是在类的 readObject 方法里面，再执行所需的对象初始化逻辑。对于一般的 Java 类来说，构造方法中包含了初始化的逻辑。可以把这些逻辑提取到一个方法中，在 readObject 方法中调用此方法。

版本更新

把一个Java对象序列化之后，所得到的字节数组一般会保存在磁盘或数据库之中。在保存完成之后，有可能原来的Java类有了更新，比如添加了额外的域。这个时候从兼容性的角度出发，要求仍然能够读取旧版本的序列化数据。在读取的过程中，当ObjectInputStream发现一个对象的定义的时候，会尝试在当前JVM中查找其Java类定义。这个查找过程不能仅根据Java类的全名来判断，因为当前JVM中可能存在名称相同，但是含义完全不同的 Java 类。这个对应关系是通过一个全局惟一标识符serialVersionUID来实现的。通过在实现了Serializable接口的类中定义该域，就声明了该Java类的一个惟一的序列化版本号。JVM会比对从字节数组中得出的类的版本号，与JVM中查找到的类的版本号是否一致，来决定两个类是否是兼容的。对于开发人员来说，需要记得的就是在实现了Serializable接口的类中定义这样的一个域，并在版本更新过程中保持该值不变。当然，如果不希望 维持这种向后兼容性，换一个版本号即可。该域的值一般是综合Java类的各个特性而计算出来的一个哈希值，可以通过Java提供的 serialver命令来生成。在Eclipse中 ,如果Java类实现了Serializable接口，Eclipse会提示并帮你生成这个serialVersionUID。

在类版本更新的过程中，某些操作会破坏向后兼容性。如果希望维持这种向后兼容性，就需要格外的注意。一般来说，在新的版本中添加东西不会产生什么问题，而去掉一些域则是不行的。

序列化安全性

前面提到，Java对象序列化之后的内容格式是 公开的 。所以可以很容易的从中提取出各种信息。从实现的角度来说，可以从不同的层次来加强序列化的安全性。

对序列化之后的流进行加密。这可以通过 CipherOutputStream来实现。

实现自己的 writeObject 和 readObject 方法，在调用 defaultWriteObject 之前，先对要序列化的域的值进行加密处理。

使用一个 SignedObject 或 SealedObject 来封装当前对象，用SignedObject 或SealedObject进行序列化。

在从流中进行反序列化的时候，可以通过ObjectInputStream的 registerValidation方法添加 ObjectInputValidation接口的实现，用来验证反序列化之后得到的对象是否合法。

RMI

RMI（Remote Method Invocation）是Java中的 远程过程调用（Remote Procedure Call，RPC）实现，是一种分布式Java应用的实现方式。它的目的在于对开发人员屏蔽横跨不同JVM和网络连接等细节，使得分布在不同JVM上的对 象像是存在于一个统一的JVM中一样，可以很方便的互相通讯。之所以在介绍对象序列化之后来介绍RMI，主要是因为对象序列化机制使得RMI非常简单。调用一个远程服务器上的方法并不是一件困难的事情。开发人员可以基于 Apache MINA或是 Netty这样的框架来写自己的网络服务器，亦或是可以采用 REST架构风格 来 编写HTTP服务。但这些解决方案中，不可回避的一个部分就是数据的编排和解排（marshal/unmarshal）。需要在Java对象和传输格式之间进行互相转换，而且这一部分逻辑是开发人员无法回避的。RMI的优势在于依靠Java序列化机制，对开发人员屏蔽了数据编排和解排的细节，要做的事情非常少。JDK 5 之后，RMI通过动态代理机制去掉了早期版本中需要通过工具进行代码生成的繁琐方式，使用起来更加简单。

RMI采用的是典型的客户端-服务器端架构。首先需要定义的是服务器端的远程接口，

这一步是设计好服务器端需要提供什么样的服务。对远程接口的要求很简单，只需要继承自RMI中的 Remote接口即可。Remote和Serializable一样，也是标记接口。远程接口中的方法需要抛出 RemoteException。定义好远程接口之后 ,实现该接口即可。如下面的Calculator是一个简单的远程接口。

```
public interface Calculator extends Remote {
    String calculate(String expr) throws RemoteException;
}
```

实现了远程接口的类的实例称为远程对象。创建出远程对象之后，需要把它注册到一个注册表之中。这是为了客户端能够找到该远程对象并调用。

```
public class CalculatorServer implements Calculator {
    public String calculate(String expr) throws RemoteException {
        return expr;
    }
    public void start() throws RemoteException, AlreadyBoundException {
        Calculator stub = (Calculator)
UnicastRemoteObject.exportObject(this, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("Calculator", stub);
    }
}
```

CalculatorServer是远程对象的Java类。在它的start方法中通过 UnicastRemoteObject的 exportObject把当前对象暴露出来，使得它可以接收来自客户端的调用请求。再通过 Registry的 rebind方法进行注册，使得客户端可以查找到。

客户端的实现就是首先从注册表中查找到远程接口的实现对象，再调用相应的方法即可。实际的调用虽然是在服务器端完成的，但是在客户端看来，这个接口中的方法就好像是在当前 JVM 中一样。这就是 RMI 的强大之处。

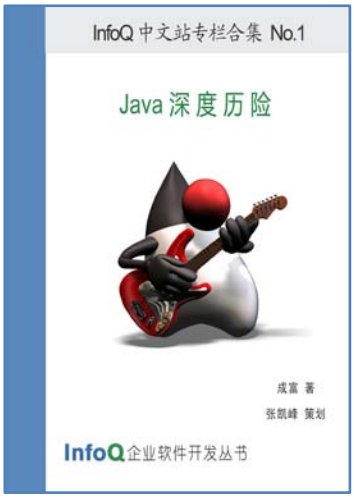
```
public class CalculatorClient {
    public void calculate(String expr) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost");
            Calculator calculator = (Calculator)
registry.lookup("Calculator");
            String result = calculator.calculate(expr);
            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在运行的时候，需要首先通过 rmiregistry 命令来启动 RMI 中用到的注册表服务器。

为了通过Java的序列化机制来进行传输，远程接口中的方法的参数和返回值，要么是Java的基本类型，要么是远程对象，要么是实现了 Serializable接口的Java类。当客户端通过RMI注册表找到一个远程接口的时候，所得到的其实是远程接口的一个动态代理对象。当客户端调用 其中的方法的时候，方法的参数对象会在序列化之后，传输到服务器端。服务器端接收到之后，进行反序列化得到参数对象。并使用这些参数对象，在服务器端调用 实际的方法。调用的返回值Java对象经过序列化之后，再发送回客户端。客户端再经过反序列化之后得到Java对象，返回给调用者。这中间的序列化过程对于使用者来说是透明的，由动态代理对象自动完成。除了序列化之外，RMI还使用了 动态类加载 技术。当需要进行反序列化的时候，如果该对象的类定义在当前JVM中没有找到，RMI会尝试从远端下载所需的类文件定义。可以在RMI程序启动的时候，通过JVM参数java.rmi.server.codebase来指定动态下载Java类文件的URL。

参考资料

- Java对象序列化规范
- RMI规范



Java 深度历险

责任编辑：张凯峰

美术编辑：胡伟红

本迷你书主页为

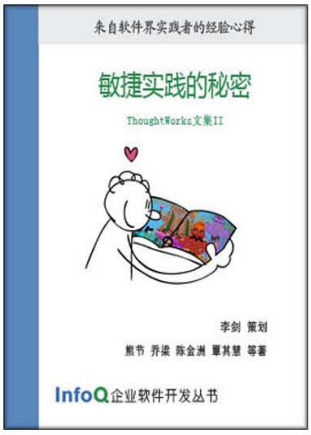
http://www.infoq.com/cn/minibooks/java-explore

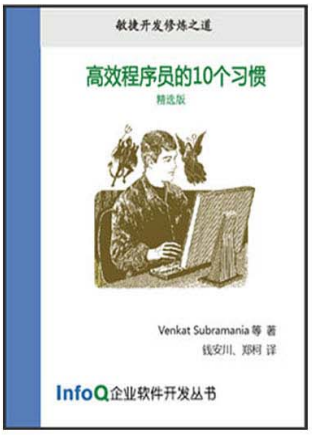
本书属于 InfoQ 企业软件开发丛书。

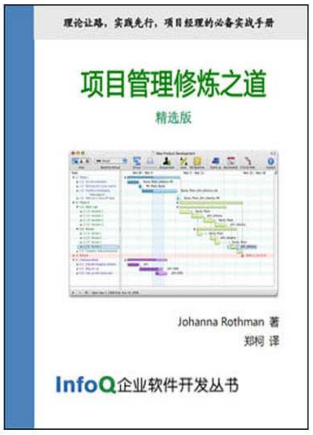
如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

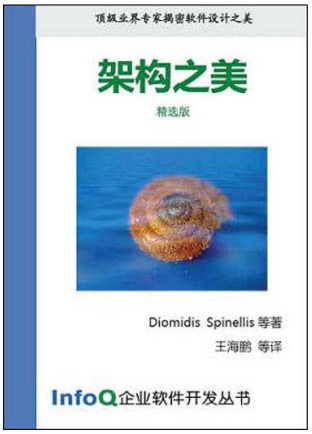
InfoQ企业软件开发丛书

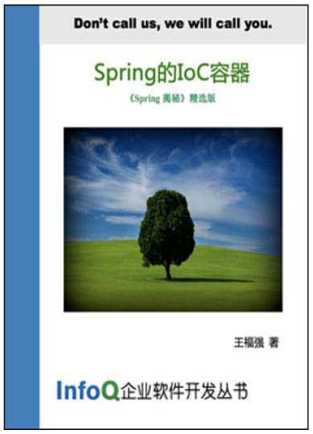
欢迎免费下载











商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com