多角度看Java中的泛型 PDF转换可能丢失图片或格式，建议阅读原文

https://www.100test.com/kao_ti2020/144/2021_2022__E5_A4_9A_E8_A7_92_E5_BA_A6_E7_c104_144837.htm 引言很多 Java 程序员都使用过集合（Collection），集合中元素的类型是多种多样的，例如，有些集合中的元素是 Byte 类型的，而有些则可能是 String 类型的，等等。Java 语言之所以支持这么多种类的集合，是因为它允许程序员构建一个元素类型为 Object 的 Collection，所以其中的元素可以是任何类型。当使用 Collection 时，我们经常要做的一件事情就是要进行类型转换，当转换成所需的类型以后，再对它们进行处理。很明显，这种设计给编程人员带来了极大的不便，同时也容易引入错误。在很多 Java 应用中，上述情况非常普遍，为了解决这个问题，使 Java 语言变得更加安全好用，近些年的一些编译器对 Java 语言进行了扩充，使 Java 语言支持了"泛型"，特别是 Sun 公司发布的 JDK 5.0 更是将泛型作为其中一个重要的特性加以推广。本文首先对泛型的基本概念和特点进行简单介绍，然后通过引入几个实例来讨论带有泛型的类，泛型中的子类型，以及范化方法和受限类型参数等重要概念。为了帮助读者更加深刻的理解并使用泛型，本文还介绍了泛型的转化，即，如何将带有泛型的 Java 程序转化成一般的没有泛型的 Java 程序。这样，读者对泛型的理解就不会仅仅局限在表面上了。考虑到多数读者仅仅是使用泛型，因此本文并未介绍泛型在编译器中的具体实现。Java 中的泛型和 C 中的模板表面上非常相似，但实际上二者还是有很大区别的，本文最后简单介绍了 Java 中的泛型与 C 模板的主要区别。泛型概览泛

型本质上是提供类型的"类型参数"，它们也被称为参数化类型（parameterized type）或参量多态（parametric polymorphism）。其实泛型思想并不是 Java 最先引入的，C 中的模板就是一个运用泛型的例子。GJ（Generic Java）是对 Java 语言的一种扩展，是一种带有参数化类型的 Java 语言。用 GJ 编写的程序看起来和普通的 Java 程序基本相同，只不过多了一些参数化的类型同时少了一些类型转换。实际上，这些 GJ 程序也是首先被转化成一般的不带泛型的 Java 程序后再进行处理的，编译器自动完成了从 Generic Java 到普通 Java 的翻译。具体的转化过程大致分为以下几个部分：将参数化类型中的类型参数"擦除"（erasure）掉；将类型变量用"上限（upper bound）"取代，通常情况下这些上限是 Object。这里的类型变量是指实例域，本地方法域，方法参数以及方法返回值中用来标记类型信息的"变量"，例如：实例域中的变量声明 A elem.，方法声明 Node (A elem){}.，其中，A 用来标记 elem 的类型，它就是类型变量。 添加类型转换并插入"桥方法"（bridge method），以便覆盖（overridden）可以正常的工作。 转化后的程序和没有引入泛型时程序员不得不手工完成转换的程序是非常一致的，具体的转化过程会在后面介绍。GJ 保持了和 Java 语言以及 Java 虚拟机很好的兼容性，下面对 GJ 的特点做一个简要的总结。类型安全。 泛型的一个主要目标就是提高 Java 程序的类型安全。使用泛型可以使编译器知道变量的类型限制，进而可以在更高程度上验证类型假设。如果没有泛型，那么类型的安全性主要由程序员来把握，这显然不如带有泛型的程序安全性高。 消除强制类型转换。泛型可以消除源代码中的许多强制类型转换，这样可以使代码更加可读，并减少

出错的机会。 向后兼容。支持泛型的 Java 编译器（例如 JDK5.0 中的 Javac）可以用来编译经过泛型扩充的 Java 程序（GJ 程序），但是现有的没有使用泛型扩充的 Java 程序仍然可以用这些编译器来编译。 层次清晰，恪守规范。无论被编译的源程序是否使用泛型扩充，编译生成的字节码均可被虚拟机接受并执行。也就是说不管编译器的输入是 GJ 程序，还是一般的 Java 程序，经过编译后的字节码都严格遵循《Java 虚拟机规范》中对字节码的要求。可见，泛型主要是在编译器层面实现的，它对于 Java 虚拟机是透明的。 性能收益。目前来讲，用 GJ 编写的代码和一般的 Java 代码在效率上是非常接近的。 但是由于泛型会给 Java 编译器和虚拟机带来更多的类型信息，因此利用这些信息对 Java 程序做进一步优化将成为可能。 以上是泛型的一些主要特点，下面通过几个相关的例子来对 Java 语言中的泛型进行说明。带有泛型的类为了帮助大家更好地理解 Java 语言中的泛型，我们在这里先来对比两段实现相同功能的 GJ 代码和 Java 代码。通过观察它们的不同点来对 Java 中的泛型有个总体的把握，首先来分析一下不带泛型的 Java 代码，程序如下：1 interface Collection {2 public void add (Object x).3 public Iterator iterator ().4 }5 6 interface Iterator {7 public Object next ().8 public boolean hasNext ().9 }10 11 class NoSuchElementException extends RuntimeException {}12 13 class LinkedList implements Collection {14 15 protected class Node {16 Object elt.17 Node next = null.18 Node (Object elt) { this.elt = elt. }19 }20 21 protected Node head = null, tail = null.22 23 public LinkedList () {}24 25 public void add (Object elt) {26 if (head == null) { head = new Node(elt). tail = head. }27 else { tail.next = new

Node(elt). tail = tail.next. }28 }29 30 public Iterator iterator () {31 32 return new Iterator () {33 protected Node ptr = head.34 public boolean hasNext () { return ptr != null. }35 public Object next () {36 if (ptr != null) {37 Object elt = ptr.elt. ptr = ptr.next. return elt.38 } else throw new NoSuchElementException ().39 }40 }.41 }42 }