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（安全性更好），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

更多资料，请加 QQ 群： 587737871

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(extends ,super)

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

更多资料，请加 QQ 群： 587737871

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：

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<? extendsNumber>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 ，因为泛型类并没有自己独立的 Class 类对象
    }
}
```

更多资料，请加 QQ 群： 587737871

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。

更多资料，请加 QQ 群： 587737871