Java泛型（二） 协变与逆变


开发者小王 (/u/6efae3f3a673) + 关注
1.3 2017.08.02 19:29* 字数 2313 阅读 4930 评论 0 喜欢 22
(/u/6efae3f3a673)

定义

逆变与协变用来描述类型转换（**type transformation**）后的继承关系，其定义：
如果A、B表示类型，f(·)表示类型转换，≤表示继承关系（比如，A≤B表示A是由B派生出来的子类）
f(·)是逆变（**contravariant**）的，当A≤B时有f(B)≤f(A)成立；
f(·)是协变（**covariant**）的，当A≤B时有f(A)≤f(B)成立；
f(·)是不变（**invariant**）的，当A≤B时上述两个式子均不成立，即f(A)与f(B)相互之间没有继承关系。

数组是协变的

Java中数组是协变的，可以向子类型的数组赋予基类型的数组引用，请看下面代码。

```
// CovariantArrays.java
class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
    public static void main(String[] args) {
        Fruit[] fruit = new Apple[10];
        fruit[0] = new Apple();
        fruit[1] = new Jonathan();
        try {
            fruit[0] = new Fruit();
        } catch (Exception e) {
            System.out.println(e);
        }
        try {
            fruit[0] = new Orange();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

main()中第一行创建了一个Apple数组，并将其赋值给一个Fruit数组引用。编译器允许你把Fruit放置到这个数组中，这对于编译器是有意义的，因为它是一个Fruit引用——它有什么理由不允许将Fruit对象或者任何从Fruit继承出来的对象（例如Orange），放置到这个数组中呢？

可能有同学会疑惑，明明Fruit[]引用的是一个Apple数组，编译器看不出来吗？还允许往里面放Fruit和Orange类的对象。你要站在编译器的角度看问题，编译器可没有人这么聪明。现代编译器大多采用的是上下文无关文法（编译器：老子归约一句是一句），符号表中存储的标识符fruit是Fruit[]类型（不然咱还怎么多态），在以后的解析过程中编译器看到fruit只会认为是Fruit[]类型。

不过，尽管编译器允许了这样做，运行时的数组机制知道它处理的是Apple[]，因此会在向数组中放置异构类型时抛出异常。程序的运行结果如下。

```
java.lang.ArrayStoreException: generics.Fruit
java.lang.ArrayStoreException: generics.Orange
```

泛型是不变的

当我们使用泛型容器来替代数组时，看看会发生什么。

```
public class NonCovariantGenerics {
    List<Fruit> flist = new ArrayList<Apple>(); // 编译错误
}
```

直接在编译时报错了。与数组不同，泛型没有内建的协变类型。这是因为数组在语言中是完全定义的，因此内建了编译期和运行时的检查，但是在使用泛型时，类型信息在编译期被擦除了（如果你不知道什么是擦除，可以去看这篇文章补补课类型擦除(https://www.jianshu.com/p/2bfbe041e6b7)），运行时也就无从检查。因此，泛型将这种错误检测移入到编译期。

通配符引入协变、逆变

协变

Java泛型是不变的，可有时需要实现协变，在两个类型之间建立某种类型的向上转型关系，怎么办呢？这时，通配符派上了用场。

```
public class GenericsAndCovariance {
    public static void main(String[] args) {
        List<? extends Fruit> flist = new ArrayList<Apple>();
        flist.add(new Apple());  // 编译错误
        flist.add(new Fruit());  // 编译错误
        flist.add(new Object());  // 编译错误
    }
}
```

现在flist的类型是<? extends Fruit>，extends指出了泛型的上界为Fruit，<? extends T>称为子类通配符，意味着某个继承自Fruit的具体类型。使用通配符可以将ArrayList<Apple>向上转型了，也就实现了协变。

然而，事情变得怪异了，观察上面代码，你再也不能往容器里放入任何东西，甚至连Apple都不行。

原因在于，List<? extends Fruit>也可以合法的指向一个List<Orange>，显然往里面放Apple、Fruit、Object都是非法的。编译器不知道List<? extends Fruit>所持有的具体类型是什么，所以一旦执行这种类型的向上转型，你就将丢失掉向其中传递任何对象的能力。

类比数组，尽管你可以把Apple[]向上转型成Fruit[]，然而往里面添加Fruit和Orange等对象都是非法的，会在运行时抛出ArrayStoreException异常。泛型把类型检查移到了编译期，协变过程丢掉了类型信息，编译器拒绝所有不安全的操作。

逆变

我们还可以走另外一条路，就是逆变。

```
public class SuperTypeWildcards {
    static void writeTo(List<? super Apple> apples) {
        apples.add(new Apple());
        apples.add(new Jonathan());
        apples.add(new Fruit());  // 编译错误
    }
}
```

我们重用了关键字super指出泛型的下界为Apple，<? super T>称为超类通配符，代表一个具体类型，而这个类型是Apple的超类。这样编译器就知道向其中添加Apple或Apple的子类型（例如Jonathan）是安全的了。但是，既然Apple是下界，那么可以知道向这样的List中添加Fruit是不安全的。

PECS

上面说的可能有点绕，那么总结下：什么使用extends，什么时候使用super。《Effective Java》给出精炼的描述：**producer-extends, consumer-super（PECS）**。

说直白点就是，从数据流来看，extends是限制数据来源的（生产者），而super是限制数据流入的（消费者）。例如上面SuperTypeWildcards类里，使用<? super Apple>就是限制add方法传入的类型必须是Apple及其子类型。

仿照上面的代码，我写了个ExtendTypeWildcards类，可以看出<? extends Apple>限制了get方法返回的类型必须是Apple及其父类型。

```
public class ExtendTypeWildcards {
    static void readFrom(List<? extends Apple> apples) {
        Apple apple = apples.get(0);
        Jonathan jonathan = apples.get(0);  // 编译错误
        Fruit fruit = apples.get(0);
    }
}
```

例子

框架和库代码中到处都是PECS，下面我们来看一些具体的例子，加深理解。

- **java.util.Collections**的**copy**方法

```
// Collections.java
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");

    if (srcSize < COPY_THRESHOLD ||
        (src instanceof RandomAccess && dest instanceof RandomAccess)) {
        for (int i=0; i<srcSize; i++)
            dest.set(i, src.get(i));
    } else {
        ListIterator<? super T> di=dest.listIterator();
        ListIterator<? extends T> si=src.listIterator();
        for (int i=0; i<srcSize; i++) {
            di.next();
            di.set(si.next());
        }
    }
}
```

copy方法限制了拷贝源src必须是T或者是它的子类，而拷贝目的地dest必须是T或者是它的父类，这样就保证了类型的合法性。

- **Rxjava**的变换

这里我们贴出一小段Rxjava2.0中map函数的源码。

```
// Observable.java
public final <R> Observable<R> map(Function<? super T, ? extends R> mapper) {
    ObjectHelper.requireNonNull(mapper, "mapper is null");
    return RxJavaPlugins.onAssembly(new ObservableMap<T, R>(this, mapper));
}
```

Function函数将<? super T>类型转变为<? extends R>类型（类似于代理模式的拦截器），可以看出extends和super分别限制输入和输出，它们可以是不同类型。

自限定的类型

理解自限定

Java泛型中，有一个好像是经常性出现的惯用法，它相当令人费解。

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

SelfBounded类接受泛型参数T，而T由一个边界类限定，这个边界就是拥有T作为其参数的SelfBounded，看起来是一种无限循环。

先给出结论：这种语法定义了一个基类，这个基类能够使用子类作为其参数、返回类型、作用域。为了理解这个含义，我们从一个简单的版本入手。

```
// BasicHolder.java
public class BasicHolder<T> {
    T element;
    void set(T arg) { element = arg; }
    T get() { return element; }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
}

// CRGWithBasicHolder.java
class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
    public static void main(String[] args) {
        Subtype st1 = new Subtype(), st2 = new Subtype();
        st1.set(st2);
        Subtype st3 = st1.get();
        st1.f();
    }
}
/* 程序输出
Subtype
*/
```

新类Subtype接受的参数和返回的值具有Subtype类型而不仅仅是基类BasicHolder类型。所以自限定类型的本质就是：基类用子类代替其参数。这意味着泛型基类变成了一种其所有子类的公共功能模版，但是在所产生的类中将使用确切类型而不是基类型。因此，Subtype中，传递给set()的参数和从get() 返回的类型都确切是Subtype。

自限定与协变

自限定类型的价值在于它们可以产生协变参数类型——方法参数类型会随子类而变化。其实自限定还可以产生协变返回类型，但是这并不重要，因为JDK1.5引入了协变返回类型。

协变返回类型

下面这段代码子类接口把基类接口的方法重写了，返回更确切的类型。

```
// CovariantReturnTypes.java
class Base {}
class Derived extends Base {}

interface OrdinaryGetter {
    Base get();
}

interface DerivedGetter extends OrdinaryGetter {
    Derived get();
}

public class CovariantReturnTypes {
    void test(DerivedGetter d) {
        Derived d2 = d.get();
    }
}
```

继承自定义类型基类的子类将产生确切的子类型作为其返回值，就像上面的get()一样。

```
// GenericsAndReturnTypes.java
interface GenericsGetter<T extends GenericsGetter<T>> {
    T get();
}

interface Getter extends GenericsGetter<Getter> {}

public class GenericsAndReturnTypes {
    void test(Getter g) {
        Getter result = g.get();
        GenericsGetter genericsGetter = g.get();
    }
}
```

协变参数类型

在非泛型代码中，参数类型不能随子类型发生变化。方法只能重载不能重写。见下面代码示例。

```
// OrdinaryArguments.java
class OrdinarySetter {
    void set(Base base) {
        System.out.println("OrdinarySetter.set(Base)");
    }
}

class DerivedSetter extends OrdinarySetter {
    void set(Derived derived) {
        System.out.println("DerivedSetter.set(Derived)");
    }
}

public class OrdinaryArguments {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedSetter ds = new DerivedSetter();
        ds.set(derived);
        ds.set(base);
    }
}
/* 程序输出
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*/
```

但是，在使用自限定类型时，在子类中只有一个方法，并且这个方法接受子类型而不是基类型为参数。

```
interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
    void set(T args);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundAndCovariantArguments {
    void testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
        s1.set(s2);
        s1.set(sbs);  // 编译错误
    }
}
```

捕获转换

<?>被称为无界通配符，无界通配符有什么作用这里不再详细说明了，理解了前面东西的同学应该能推断出来。无界通配符还有一个特殊的作用，如果向一个使用<?>的方法传递原生类型，那么对编译期来说，可能会推断出实际的参数类型，使得这个方法可以回转并调用另一个使用这个确切类型的方法。这种技术被称为捕获转换。下面代码演示了这种技术。

```
public class CaptureConversion {
    static <T> void f1(Holder<T> holder) {
        T t = holder.get();
        System.out.println(t.getClass().getSimpleName());
    }
    static void f2(Holder<?> holder) {
        f1(holder);
    }
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Holder raw = new Holder<Integer>(1);
        f2(raw);
        Holder rawBasic = new Holder();
        rawBasic.set(new Object());
        f2(rawBasic);
        Holder<?> wildcarded = new Holder<Double>(1.0);
        f2(wildcarded);
    }
}
/* 程序输出
Integer
Object
Double
*/
```

捕获转换只有在这样的情况下可以工作：即在方法内部，你需要使用确切的类型。注意，不能从f2()中返回T，因为T对于f2()来说是未知的。捕获转换十分有趣，但是非常受限。