看下面一段代码：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. // public final class Integer extends Number
2. Number num = **new** Integer(1);
3. List<Number> list = **new** ArrayList<>();
4. list.add(**new** Integer(3));
5. ArrayList<Number> list = **new** ArrayList<Integer>(); //type mismatch
7. List<? **extends** Number> list = **new** ArrayList<Number>();
8. list.add(**new** Integer(1)); //error

为什么Number的对象可以由Integer实例化，而ArrayList<Number>的对象却不能由ArrayList<Integer>实例化？list中的<? extends Number>声明其元素是Number或Number的派生类，为什么不能add Integer?为了解决这些问题，需要了解Java中的逆变和协变以及泛型中通配符用法。

**1. 逆变与协变**

       Java中String类型是继承自Object的，姑且记做String ≦ Object，表示String是Object的子类型，String的对象可以赋给Object的对象。而Object的数组类型Object[]，理解成是由Object构造出来的一种新的类型,可以认为是一种构造类型，记f(Object)，那么可以这么来描述协变和逆变：  
       当A ≦ B时,如果有f(A) ≦ f(B),那么f叫做协变；  
       当A ≦ B时,如果有f(B) ≦ f(A),那么f叫做逆变；  
       如果上面两种关系都不成立则叫做不可变。

**2. 泛型中的通配符实现协变与逆变**

       JAVA中泛型是不变的，可有时需要实现逆变与协变，怎么办呢？这时就需要通配符?。  
       <? extends>实现了泛型的协变，比如：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. List<? **extends** Number> list = **new** ArrayList<>();

       “? extends Number”则表示通配符”?”的上界为Number，换句话说就是，“? extends Number”可以代表Number或其子类，但代表不了Number的父类（如Object），因为通配符的上界是Number。  
       于是有“? extends Number” ≦ Number，则List<? extends Number> ≦ List< Number >。那么就有：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. List<? **extends** Number> list001 = **new** ArrayList<Integer>();
2. List<? **extends** Number> list002 = **new** ArrayList<Float>();

       但是这里不能向list001、list002添加除null以外的任意对象。可以这样理解一下，List<Integer>可以添加Interger及其子类，List<Float>可以添加Float及其子类，List<Integer>、List<Float>都是List<? extends Animal>的子类型，如果能将Float的子类添加到List<? extends Animal>中，就说明Float的子类也是可以添加到List<Integer>中的，显然是不可行。故java为了保护其类型一致，禁止向List<? extends Number>添加任意对象，不过却可以添加null。  
  
       <? super>实现了泛型的逆变，比如：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. List<? **super** Number> list = **new** ArrayList<>();

       “? super Number” 则表示通配符”?”的下界为Number。为了保护类型的一致性，因为“？ super Number”可以是Object或其他Number的父类，因无法确定其类型，也就不能往List<? super Number >添加Number的任意父类对象。但是可以向List<? super Number >添加Number及其子类。

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. List<? **super** Number> list001 = **new** ArrayList<Number>();
2. List<? **super** Number> list002 = **new** ArrayList<Object>();
3. list001.add(**new** Integer(3));
4. list002.add(**new** Integer(3));

**3．PECS**

       现在问题来了：究竟什么时候用extends什么时候用super呢？《Effective Java》给出了答案：  
       PECS: producer-extends, consumer-super.  
       比如，一个简单的Stack API：

1. **public** **class** Stack<E>{
2. **public** Stack();
3. **public** **void** push(E e):
4. **public** E pop();
5. **public** **boolean** isEmpty();
6. }

       要实现pushAll(Iterable<E> src)方法，将src的元素逐一入栈：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. **public** **void** pushAll(Iterable<E> src){
2. **for**(E e : src)
3. push(e)
4. }

       假设有一个实例化Stack<Number>的对象stack，src有Iterable<Integer>与 Iterable<Float>；在调用pushAll方法时会发生type mismatch错误，因为Java中泛型是不可变的，Iterable<Integer>与 Iterable<Float>都不是Iterable<Number>的子类型。因此，应改为

1. // Wildcard type for parameter that serves as an E producer
2. **public** **void** pushAll(Iterable<? **extends** E> src) {
3. **for** (E e : src)
4. push(e);
5. }

       要实现popAll(Collection<E> dst)方法，将Stack中的元素依次取出add到dst中，如果不用通配符实现：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. // popAll method without wildcard type - deficient!
2. **public** **void** popAll(Collection<E> dst) {
3. **while** (!isEmpty())
4. dst.add(pop());
5. }

       同样地，假设有一个实例化Stack<Number>的对象stack，dst为Collection<Object>；调用popAll方法是会发生type mismatch错误，因为Collection<Object>不是Collection<Number>的子类型。因而，应改为：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. // Wildcard type for parameter that serves as an E consumer
2. **public** **void** popAll(Collection<? **super** E> dst) {
3. **while** (!isEmpty())
4. dst.add(pop());
5. }

       在上述例子中，在调用pushAll方法时生产了E 实例（produces E instances），在调用popAll方法时dst消费了E 实例（consumes E instances）。Naftalin与Wadler将PECS称为Get and Put Principle。  
  
       java.util.Collections的copy方法(JDK1.7)完美地诠释了PECS：

**[java]** [view plain](http://blog.csdn.net/zero__007/article/details/52245475) [copy](http://blog.csdn.net/zero__007/article/details/52245475)

1. **public** **static** <T> **void** copy(List<? **super** T> dest, List<? **extends** T> src) {
2. **int** srcSize = src.size();
3. **if** (srcSize > dest.size())
4. **throw** **new** IndexOutOfBoundsException("Source does not fit in dest");
6. **if** (srcSize < COPY\_THRESHOLD ||
7. (src **instanceof** RandomAccess && dest **instanceof** RandomAccess)) {
8. **for** (**int** i=0; i<srcSize; i++)
9. dest.set(i, src.get(i));
10. } **else** {
11. ListIterator<? **super** T> di=dest.listIterator();
12. ListIterator<? **extends** T> si=src.listIterator();
13. **for** (**int** i=0; i<srcSize; i++) {
14. di.next();
15. di.set(si.next());
16. }
17. }
18. }

## 实验准备

　　现在在上篇文章展示的Decorator类型的基础上，增加一些代码，如代码清单1所示。

代码清单1

**[java]** [view plain](http://blog.csdn.net/beliefer/article/details/50894265) [copy](http://blog.csdn.net/beliefer/article/details/50894265)

1. /\*\*
2. \*
3. \* 描 述：Exp2使用br/>
4. \* 作 者：jiaan.gja<br/>
5. \* 历 史: (版本) 作者 时间 注释 <br/>
6. \* @param itemList
7. \*/
8. **public** **void** doDecorate2(List<? **extends** T> itemList) {
9. **for**(**int** i = 0; i < itemList.size(); i++) {
10. System.out.println(itemList.get(i));
11. }
12. }
14. /\*\*
15. \*
16. \* 描 述：Exp2使用<br/>
17. \* 作 者：jiaan.gja<br/>
18. \* 历 史: (版本) 作者 时间 注释 <br/>
19. \* @param itemList
20. \* @param t
21. \*/
22. **public** **void** addDecorate2(List<? **extends** T> itemList, T t) {
23. // itemList.add(t);
24. }

可以看到这跟[《Java泛型的基本使用》](http://blog.csdn.net/beliefer/article/details/50763741" \t "_blank)中的doDecorate和addDecorate方法几乎一样，唯一的区别是将List<T>改为List<? extends T>，这种语法被称为协变。如果你已经在IDE里写了上面的代码，肯定发现addDecorate2中的语句itemList.add(t);已经编译报错。所以我们先将itemList.add(t);这条语句暂时注释掉，先来看看doDecorate2有没有达到我们的预期。

## 实验：泛型协变

　　现在我们尝试下协变的用途，即调用doDecorate2方法，如代码清单2所示。

代码清单2

**[java]** [view plain](http://blog.csdn.net/beliefer/article/details/50894265) [copy](http://blog.csdn.net/beliefer/article/details/50894265)

1. /\*\*
2. \*
3. \* 类 名: Exp2<br/>
4. \* 描 述: 泛型的协变使用<br/>
5. \* 作 者: jiaan.gja<br/>
6. \* 创 建： 2015年8月20日<br/>
7. \*
8. \* 历 史: (版本) 作者 时间 注释
9. \*/
10. **class** Exp2 {
11. **public** **static** **void** main(String[] args) {
13. Decorator<Auction> decoratorA = **new** Decorator<Auction>();
14. List<Auction> listA = **new** ArrayList<Auction>();
15. Auction auctionOne = **new** Auction("auctionOne");
16. Auction auctionTwo = **new** Auction("auctionTwo");
17. decoratorA.addDecorate2(listA, auctionOne);
18. decoratorA.addDecorate2(listA, auctionTwo);
19. decoratorA.doDecorate2(listA);
21. Decorator<Table> decoratorB = **new** Decorator<Table>();
22. List<Table> listB = **new** ArrayList<Table>();
23. Table tableOne = **new** Table("tableOne", 10);
24. Table tableTwo = **new** Table("tableTwo", 20);
25. decoratorB.addDecorate2(listB, tableOne);
26. decoratorB.addDecorate2(listB, tableTwo);
27. decoratorB.doDecorate2(listB);
29. Decorator<Service> decoratorC = **new** Decorator<Service>();
30. List<Service> listC = **new** ArrayList<Service>();
31. Service serviceOne = **new** Service("serviceOne", "methodOne");
32. Service serviceTwo = **new** Service("serviceTwo", "methodTwo");
33. decoratorC.addDecorate2(listC, serviceOne);
34. decoratorC.addDecorate2(listC, serviceTwo);
35. decoratorC.doDecorate2(listC);
37. /\*
38. \* 协变测试开始
39. \*/
40. decoratorA.doDecorate2(listB);
41. decoratorA.doDecorate2(listC);
43. }
44. }

首先，我们看到doDecorate2方法至少能达到[《Java泛型的基本使用》](http://www.cnblogs.com/jiaan-geng/p/4846134.html" \t "_blank)中的doDecorate方法的效果。由于声明了decoratorA的类型是Decorator<Auction>，那么此时实例decoratorA的doDecorate2的参数可以认为是下面这样：

List<? extends Auction> itemList

而listB的类型是ArrayList<Table>，listC的类型是ArrayList<Service>，也就是说协变允许：

List<? extends Auction> itemList = new ArrayList<Auction>();

List<? extends Auction> itemList = new ArrayList<Table>();

List<? extends Auction> itemList = new ArrayList<Service>();

这说明itemList中既可以存储Auction，也可以存储Table和Service。我们做个假设，当你想从itemList中获取Table的时候，Java编译器怀疑这实际可能是个Service，因此会有编译错误。如果想从itemList中获取Auction是没有问题的，因为无论是Table还是Service它们都可以被自动转换为父类Auction。这些推测，可以通过代码清单3验证。

代码清单3

**[java]** [view plain](http://blog.csdn.net/beliefer/article/details/50894265) [copy](http://blog.csdn.net/beliefer/article/details/50894265)

1. List<? **extends** Auction> itemList = listA;
2. Auction a = itemList.get(0);
3. Table t = itemList.get(0);
4. Service s = itemList.get(0);
5. itemList = listB;
6. a = itemList.get(0);
7. t = itemList.get(0);
8. s = itemList.get(0);
9. itemList = listC;
10. a = itemList.get(0);
11. t = itemList.get(0);
12. s = itemList.get(0);

同样的道理，当你想要向itemList中添加Table时，编译器怀疑itemList实际是个ArrayList<Service>，因此是不允许的。在获取Auction时是没有问题的，那么添加Auction会不会也可以？由于itemList可能是ArrayList<Service>或者ArrayList<Table>，根据上一篇[《Java泛型的基本使用》](http://blog.csdn.net/beliefer/article/details/50763741" \t "_blank)的结论，这实际也是不可以的。可以用下列代码验证：

**[java]** [view plain](http://blog.csdn.net/beliefer/article/details/50894265) [copy](http://blog.csdn.net/beliefer/article/details/50894265)

1. Auction auctionThree = **new** Auction("auctionThree");
2. itemList.add(auctionThree);
3. Auction tableThree = **new** Auction("tableThree");
4. itemList.add(tableThree);
5. Auction serviceThree = **new** Auction("serviceThree");
6. itemList.add(serviceThree);

因此这也说明了刚开始看到的addDecorate2中的语句itemList.add(t);为什么会编译报错。虽然如此，我们依然希望能将子类型添加到父类型声明的泛型中，这该怎么办？[**Java**](http://lib.csdn.net/base/java)当然也考虑到这个问题，请看下一篇[《Java泛型的逆变》](http://blog.csdn.net/beliefer/article/details/50953198)。

## 总结

　　如果从泛型中获取子类，应该使用协变。