`Collection`接口

`Iterable`接口

先介绍`Iterable`接口

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) { // 经常使用的东西
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

这就是`Iterator`接口需要实现的方法（有一个默认实现）

涉及到了设计模式，这里的`Consumer`是一个有`accept`方法的接口，属于消费者模式

`Collection`接口

直接看源码

```
public interface Collection<E> extends Iterable<E>{...}
```

也就是，所有的实现了`Collection`的类都有`Collection`接口的一些方法，另外由于`Collection`继承自`Iterable`接口，所以也需要有`Iterable`相关的方法

`Collection`的方法选讲

直接看源码：

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
```

```
    default <T> T[] toArray(IntFunction<T[]> generator) {
        return toArray(generator.apply(0));
    }
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
            }
        }
        return removed;
    }
    boolean retainAll(Collection<?> c); // 取当前集合与c的交集
    void clear();
}
```

这里基本上看一眼都能理解是干嘛的

实际上有些范型设计上需要考虑的坑，我们暂时不管它，知道怎么用就行

`Iterator`接口

直接看代码：

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

`List`

这里主要关注设计的思想，也就是为什么这么设计

`List`是继承自`Collection`的抽象类

如果我们需要一个`List`，类型可以指定成`List`但是`new`后面必须跟一个实现过的类，如下：

```
List<String>ls=new ArrayList<>(); // new ArrayList<String>()也行，这里实际上进行了自动类型推断
```

除了`List`之外，其他的`Collection`，比如`Map`，`Set`都是接口

迭代器、`foreach`和`forEach`

迭代器

使用方式

Java的迭代器和C++的不太一样：

```
List<String>ls=new ArrayList<String>(); // 可以写成List ls=...
ls.add("hello");ls.add("world");ls.add("!");
System.out.println(ls); // [hello, world, !]
Iterator<String>iter=ls.iterator();
while(iter.hasNext()){
    System.out.println(iter.next());
}
```

C++里需要`auto iter=ls.begin();`，遍历尽的标识：`iter!=ls.end()`

Java的方式见上（顺便，py似乎也是这么实现的）

源码-设计原理

先看`Iterator`接口

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() { // Java支持直接通过迭代器来删除内容233
        throw new UnsupportedOperationException("remove"); // 默认实现的是抛出一个“方法不被支持”的异常
    }
    default void forEachRemaining(Consumer<? super E> action) { // 默认实现就是正常的实现
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

这里的迭代器设计很奇怪：并没有一个方法返回迭代器现在指向的空间内容是什么，而只有一个`next()`用来同时返回内容和迭代

另外，我们看看`ArrayList`的相关源码：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    public Iterator<E> iterator() {
        return new Itr();
    }
}
```

这里有一个`Itr`，我们再看看`Itr`：

```
// in List, 也就是Itr是一个内部类
private class Itr implements Iterator<E> { // 也就是迭代器也是一种接口
    int cursor;       // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;

    Itr() {}

    public boolean hasNext() {
        return cursor != size;
    }

    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
```

```
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }
}
```

这里涉及到了内部类的东西。。。总之就是，Java通过一种内部类来实现`iterator`，这样`iterator`天然具有对于产生`iterator`的容器的访问权限，从而可以进行方便的迭代设计

`remove`方法

迭代器有一个`remove`方法，我们简单了解一下这是干嘛的

C++里面，有一种“尾后迭代器”的概念，比如`vec.end()`，用来表示已经达到容器的结尾。但是对于Java来说不一样，Java的迭代器是从“头前”开始的，所以`Iterator<StringBuffer>sb=lsb.iterator();`之后，先`itr.next();`，然后才能`itr.remove()`。顺便，`remove`之后`itr`就会指向下一个元素

```
List<StringBuffer>lsb=new ArrayList<>();
lsb.add(new StringBuffer("hello"));
lsb.add(new StringBuffer("world"));
lsb.add(new StringBuffer("!"));
Iterator<StringBuffer>sb=lsb.iterator();
// sb.remove(); // WRONG!必须先next()一次
sb.next();
sb.remove();
System.out.println(sb.next());
System.out.println(lsb);
```

使用注意

和C++一样的是：使用迭代器的时候，不能改变容器的长度，否则会出现错误。具体来说如下：

```
System.out.println(ls); // [hello, world, !]
Iterator<String>iter=ls.iterator();
int i=0;
while(iter.hasNext()){
    System.out.println(iter.next());
    if((i%2)==1){
        ls.add(String.valueOf(i));
    }
    ++i;
}
```

产生运行时异常（使用`foreach`的访问同理）

`foreach`

使用方式

和C++不一样的是，Java并不能提供C++里`for((int*) & x:vec)`的方式来提供一个指针的引用，但是并不意味着不能通过`foreach`改变容器里的元素的值：

```
List<StringBuffer>lsb=new ArrayList<>();
lsb.add(new StringBuffer("hello"));
lsb.add(new StringBuffer("world"));
lsb.add(new StringBuffer("!"));
for(StringBuffer sb:lsb){ // 这里的sb就是引用！当然，如果lsb的E是String的话，就不能使用
sb.append了，因为String没有append
    sb.append('.'); // 改变了lsb里面每个元素的值
}
for(var sb:lsb){
    System.out.println(sb);
}
/*
hello.
world.
!.
*/
```

实现原理

没有查到相关资料，据说是实现了`Iterator`接口所以可以遍历

`forEach`

实际上`forEach`是`Iterable`里面要求的，所以所有继承自`Collection`的容器都会有`forEach`方法

一般的`forEach`写法：

```
Set<Integer>si=new HashSet<>(){
    {
        for(int i=1;i<=10;++i){
            add(i);
        }
    }
};
si.forEach(new Consumer<Integer>() {
```

```
    @Override
    public void accept(Integer integer) {
        System.out.println(integer);
    }
});
si.forEach((i)->{System.out.println(i);}}; // lambda表达式的方式
```

看一眼就知道咋弄了

`List`存的是引用

我们看下面的代码：

```
List<StringBuffer>lsb=new ArrayList<>();
StringBuffer sb=new StringBuffer("hello");
lsb.add(sb);lsb.add(new StringBuffer("world"));lsb.add(new StringBuffer("!"));
sb.append("ooo");
System.out.println(lsb);
```

输出为：`[helloooo, world, !]`

将`sb`放入`List`，然后在外部调用`sb`的`.append()`方法，输出`List`存储的发现也改变了，可以得到结论：`List`实际上存储的是引用！

`Queue`

`Queue`也是继承自`Collection`的抽象类，并且`Queue`有一系列奇怪的继承结构。。。这里只介绍一种`Queue`：`LinkedList`及其方法

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable // 这个实际上是一个链表。。。有毒
{
    transient int size = 0;
    transient Node<E> first;
    transient Node<E> last;
    public LinkedList() {
    }
    public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
    }
    public int size() {
        return size;
```

```
    }

    /**
     * Appends the specified element to the end of this list.
     */
    public boolean add(E e) {
        linkLast(e);
        return true;
    }


    /**
     * Retrieves, but does not remove, the head (first element) of this list.
     */
    public E peek() {
        final Node<E> f = first;
        return (f == null) ? null : f.item;
    }


    /**
     * Retrieves and removes the head (first element) of this list.
     */
    public E poll() {
        final Node<E> f = first;
        return (f == null) ? null : unlinkFirst(f);
    }


    /**
     * Returns true if this collection contains no elements.
     */
    public boolean isEmpty() {
        return size() == 0;
    }


    /**
     * Appends all of the elements in the specified collection to the end of
     * this list, in the order that they are returned by the specified
     * collection's iterator.
     */
    public boolean addAll(Collection<? extends E> c) {
        return addAll(size, c);
    }
}
```

这玩意实际上是一个链表，所以其实还有很多方法，但是如果把它当队列的话，就这些是需要用到的

Map

Map并没有继承自任何接口！

和List或者Set继承自Collection接口不同，Map没有继承自任何接口！所以需要专门讲讲它

Map方法选讲

Entry内部接口

由于Map没有实现Iterable接口，也就没有有Iterator内部类的必要（当然，实际上来说，Iterator只允许一个范型类型，Map有两个<键和值>，也用不成）

但是我们还是希望有一个迭代器存在的，这就是Entry的作用了：

```
// in Map
interface Entry<K, V> {
    K getKey();
    V getValue();
    V setValue(V value);
    boolean equals(Object o);
    int hashCode();
}
```

可以看出Entry的这几个方法，基本都是为了迭代器设计的

Map方法

注意下，Map的forEach方法表现了Map的迭代器应该怎么写，自己写的时候可以参照

replaceAll则表现了如果想对每一个Map的key改变value应该怎么写

```
public interface Map<K, V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    /*
     * The effect of this call is equivalent to that
```

```
     * of calling {put(k, v)} on this map once
     * for each mapping from key {k} to value {v} in the
     * specified map.
    */
    void clear();
    Set<K> keySet(); // 返回Key的Set
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet(); // 这里返回的是Set，所以entrySet().iterator()返回的就是Iterator<Map.Entry<K,V>>
    default V getOrDefault(Object key, V defaultValue) {
        V v;
        return (((v = get(key)) != null) || containsKey(key))
            ? v
            : defaultValue;
    }
    default void forEach(BiConsumer<? super K, ? super V> action) {
        Objects.requireNonNull(action);
        for (Map.Entry<K, V> entry : entrySet()) { // 这就是Map的迭代器写法
            K k;
            V v;
            try {
                k = entry.getKey();
                v = entry.getValue();
            } catch (IllegalStateException ise) {
                // this usually means the entry is no longer in the map.
                throw new ConcurrentModificationException(ise);
            }
            action.accept(k, v);
        }
    }
    default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {
        Objects.requireNonNull(function);
        for (Map.Entry<K, V> entry : entrySet()) { // 对每一个key改变其value的写法
            K k;
            V v;
            try {
                k = entry.getKey();
                v = entry.getValue();
            } catch (IllegalStateException ise) {
                // this usually means the entry is no longer in the map.
                throw new ConcurrentModificationException(ise);
            }

            // ise thrown from function is not a cme.
            v = function.apply(k, v);
            try {
                entry.setValue(v);
            } catch (IllegalStateException ise) {
                // this usually means the entry is no longer in the map.
```

```
                throw new ConcurrentModificationException(ise);
            }
        }
    }
    default boolean remove(Object key, Object value) {
        Object curValue = get(key);
        if (!Objects.equals(curValue, value) ||
            (curValue == null && !containsKey(key))) {
            return false;
        }
        remove(key);
        return true;
    }
    default boolean replace(K key, V oldValue, V newValue) {
        Object curValue = get(key);
        if (!Objects.equals(curValue, oldValue) ||
            (curValue == null && !containsKey(key))) {
            return false;
        }
        put(key, newValue);
        return true;
    }
    default V replace(K key, V value) {
        V curValue;
        if (((curValue = get(key)) != null) || containsKey(key)) {
            curValue = put(key, value);
        }
        return curValue;
    }
}
```

基本上看下实现都可以理解

需要注意的是`forEach`的使用，我们后面会讲

迭代器、`foreach`和`forEach`

首先明确一点，由于`Map`不是`Iterable`->`Collection`一条线下来的继承，所以实现是不一样的，这也是为啥需要专门讲`Map`的遍历的原因

迭代器

`Map`没有继承自`Iterator`迭代器，但是有`Entry`可以做和迭代器差不多的功能：

```
Map<String, Double> map = new HashMap<>() {
    {
        put("a", 0.1);put("b", 0.2);put("c", 0.3);
    }
};
Iterator< Map.Entry<String,Double> >itr=map.entrySet().iterator();
while(itr.hasNext()){
    Map.Entry<String,Double>entry=itr.next();
    entry.setValue(entry.getValue()+10);
    System.out.println(entry.getKey()+" , "+entry.getValue());
}
```

这就是`Map`迭代器的工作方式

`foreach`

和迭代器一样，`Map`的`foreach`本质也是对`entrySet()`的遍历

```
for (Map.Entry<String, Double> e : map.entrySet()) {
    e.setValue(e.getValue() + 10);
}
```

（语句和上面等价）

`forEach`

`Map`自己实现了一个`forEach`，用法如下：

```
map.forEach(new BiConsumer<String, Double>() {
    @Override
    public void accept(String s, Double aDouble) {
        System.out.println(s + " : " + aDouble);
    }
});
```

`forEach`接受的是一个实现了`BiConsumer`接口的类。`BiConsumer`接口如下：

```
public interface BiConsumer<T, U> {
    void accept(T t, U u);
    default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {
        Objects.requireNonNull(after);

        return (l, r) -> {
            accept(l, r);
            after.accept(l, r);
        };
    }
}
```

一些Map实现的行为

我们常用的`Map`主要是`HashMap`和`TreeMap`，两者的不同在查找`key`的方式不同

`HashMap`会使用`Key`的`hashCode`方法生成一个哈希值，并根据哈希值构建哈希表；`TreeMap`则使用`Key`的`.compare`或者传入的`comparator`表示元素的优先级建立红黑树

主要需要注意的是，**`Object`的`hashCode`方法返回的是地址**，所以下面的代码：

```
public class Main {
    public static void main(String[] args) {
        Map<Tes, Integer> map = new HashMap<>();
        map.put(new Tes(1),1);
        System.out.println(map.getOrDefault(new Tes(1),-1));
    }
    static class Tes{
        int id;
        Tes(int id){
            this.id=id;
        }
    }
}
```

`getOrDefault`里面的`Tes`地址和`put`里面的并不一样，即两者的`hashCode`不一样，所以输出的是`-1`

另外`hashCode`没有办法保证元素性质，元素可能是乱序的（即使用`Entry`之类的遍历器时遍历顺序不一定有序）

范型类

一个和C++明显的机制上的不同

C++的范型在一定程度上可以简单理解成宏定义直接替换，比如`template<typename T>class CLS{}`，如果我们有了`CLS<int>,CLS<double>,CLS<long long>`，那么实际上就会编译出三份代码

而Java的范型使用的是使用`Object`或者特定类替换的技术

后面会解释

声明

范型类

比如我们写一个`Pair`类

```
import java.util.Objects;

public class Pair<Tf extends Comparable<Tf> & Cloneable,Ts extends Comparable<Ts> &
Cloneable>
        implements Comparable<Pair<Tf,Ts>>{
    // 范型的类使用任何字符都行，不一定用T,S,K,V之类的
    public Tf first;
    public Ts second;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Pair)) return false;
        Pair<?, ?> pair = (Pair<?, ?>) o; // 这里有个坑：为啥要这么写。。。
        return Objects.equals(first, pair.first) && Objects.equals(second,
pair.second);
    }

    @Override
    public int hashCode() {
        return Objects.hash(first, second);
    }

    @Override
    public String toString() {
        return first + " "+second;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
        // 这个实现是错误的。这里的正确实现必须使用反射
        // Cloneable接口并不强制要求有clone方法。。。有毒
    }
```

```
    @Override
    public int compareTo(Pair<Tf, Ts> o) {
        if(first.compareTo(o.first)!=0){
            return first.compareTo(o.first);
        }else{
            return second.compareTo(o.second);
        }
    }
}
```

这里顺便写了一些常见的`Object`方法的重载形式，顺便加上了通配符、范型接口的内容

范型方法

我们也可以（像C++一样）实现范型方法：

```
public static <T> T getMiddle(T[] a){
    return a[a.length/2];
}
```

调用方法：

```
Integer[] ar=new Integer[10];
Integer x=Tes.<Integer>getMiddle(ar); // OK
Integer y=getMiddle(ar); // OK
// Integer z=<Integer>getMiddle(ar); // WRONG!
```

Java的范型有时候会推断类型，这里的`y`就使用了类型推断

类型推断是有可能出问题的，遇到具体问题具体对待，总之建议不要依赖类型推断

类型变量限定

形式

在进行范型的设计的时候，我们有时候希望对象拥有一些方法，比如（例子来自《Java核心技术 卷I》）：

```
public static <T> T min(T[] a){
    if(a==null || a.length==0){
        return null;
    }
    T smallest=a[0];
    for(int i=1;i<a.length;++i) if (smallest.compareTo(a[i])>0) smallest=a[i];
    return smallest;
}
```

我们希望`T`有`compareTo`方法，可以很简单地改变第一行为下面语句实现：

```
public static <T extends Comparable> T min(T[] a){...}
```

`Comparable`是一个接口，也就是在限定类型变量的时候，无论表示“实现接口”还是“继承自基类”，都需要使用`extends`

也可以添加多个限定，此时必须将“继承自基类”放在第一个，“实现接口”顺序随意，之间使用`&`分隔

这种对范型类的限制在C++里面，直到C++20的`concept`和`requires`技术出现前，似乎都不太可能实现

与“类型擦除”相关

在没有限定的时候，Java的范型相当于使用`Object`类替代`T`，比如：

```
class Tes<T> {
    private T t;
    public T gett(){
        return t;
    }
}
// 在Java虚拟机中：
class Tes{
    private Object t;
    public Object gett(){
        return t;
    }
}
```

这只是在虚拟机的实现，实际上还有运行时自动类型转换之类的事，比如`Tes<Integer>ti=new Tes<>();int x=ti.gett();`，这里不需要对`gett`的返回值类型转换，Java虚拟机会自动执行

对于有限定的类，会使用第一个限定作为类的里面的类型：

```
public class Pair<Tf extends Comparable & Cloneable,Ts extends Comparable & Cloneable>{
    // 顺便，为了简化内容，这里和上面不同的是，我们将Comparable<T>替换成了Comparable
    public Tf first;
    public Ts second;
}
// 在虚拟机里
public class Pair{
    public Comparable first;
    public Comparable second;
}
```

这样在运行时，需要用到其他接口/类继承下来的方法时，会进行自动类型转换

所以为了效率考虑，tagging接口（也就是没有实现方法的接口）不应该放在第一个；最好将使用最多的方法所在接口/类放在第一个，以减少自动类型转换

但是这个有一个问题，就是`instanceof`无法检测范型类型

这里只是简单介绍了“类型擦除”在Java虚拟机中的形式，之后会逐渐深入这个Java范型实现机制

类型擦除的一些定义

为了后面能够更好地说明，我们需要一些严格定义。以下面的代码为例：

```
class Tes<K extends Comparable,V>{
    V get(K key);
}
// 类型擦除后
class Tes{
    Object get(Comparable);
}
```

我们管`K`和`V`在类型擦除后的“替代品”`Comparable`和`Object`称为“原始类型”，将`K`和`V`称为范型类型

在类型擦除后，编译器需要进行自动类型转换，比如`Tes t=new Tes();V v=t.get(key);`，这里就涉及了两个强制类型转换：`key`从`K`转成`Comparable`，`get`返回的`Object`转化成`V`。这种类型转换是很多的，我们统称为自动强制类型转换

范型接口和“实现范型接口”

我们看`Comparable`：

```
public interface Comparable<T>{
    public int compareTo(T o);
}
```

这就是`Comparable`的全部内容

我们不妨看下类型擦除之后是什么样的：

```
// 类型擦除后
public interface Comparable{
    public int compareTo(Object o);
}
```

如果我们每次使用都强制类型转换，这个接口设计其实是没有问题的，也就是

```
class Tes1 implements Comparable{...}
class Tes2 implements Comparable<Tes2>{...}
```

两个都是语法上可行的，`Tes1`要求我们每次都强制类型转换，`Tes2`则是Java虚拟机帮助我们进行自动强制类型转换

不过我们自己实现的时候，最好写`implements Comparable<Tes2>`的形式

通配符

范型类实例的继承关系

我们需要先明确范型类实例的继承关系，这有助于让我们理解为何需要通配符设计

我们知道`Integer`是继承自`Numebr`的一个类，那么我们就有一个问题：`List<Integer>`可以看作继承自`List<Number>`吗？答案是不行，我们做个实验：

```
List<Number> numberList=new LinkedList<>();
List<Integer>integerList=new LinkedList<>();
// System.out.println(integerList instanceof List<Number>); // CE，因为List<Number>就是不合法的形式
// numberList=integerList; // CE，因为：
// 不兼容的类型：java.util.List<java.lang.Integer>无法转换为java.util.List<java.lang.Number>
```

如果`List<Integer>`可以看作继承自`List<Number>`，那么这个赋值是完全没问题的。但是这个赋值过不了编译

实际上，在《Java核心技术 卷I 》里面就明确提出了，`List<Integer>`和`List<Numebr>`是两个完全没有继承关系的类

但是我们不喜欢这样，比如下面的代码：

```
public class Wildcard {
    public static void main(String[] args) {
        List<Number>list=getListNumber();
//      list=getListInteger(); // #
    }
    static List<Number>getListNumber(){...}
    static List<Integer>getListInteger(){...}
}
```

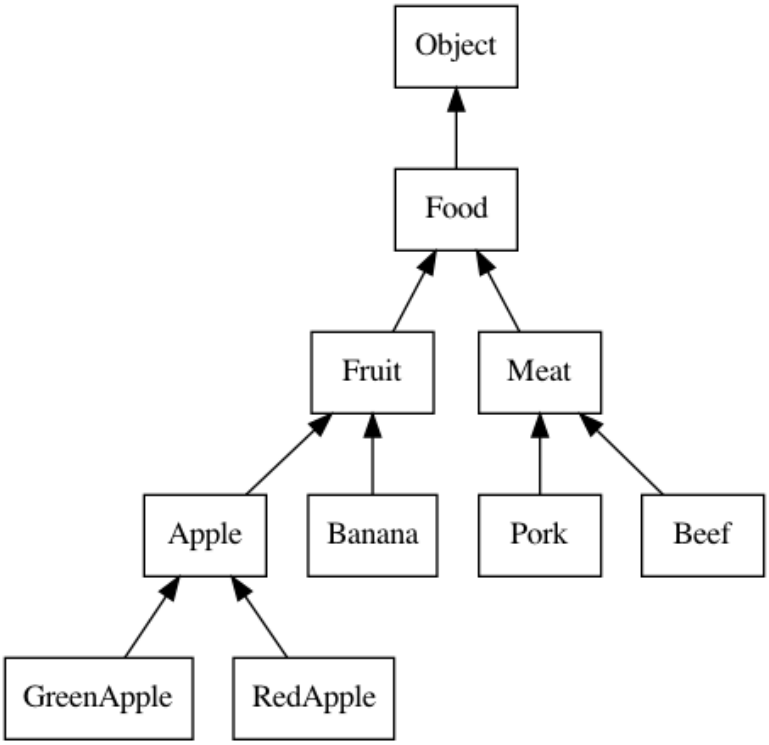
我们希望`#`那行代码能跑起来，这样就不用重新新建一个变量专门存`getListInteger`返回的东西，当然最好还能保持它的`List`性质。怎么实现呢？通配符！

> 顺便，有一个Java数组和范型的区别，下面的行为没有任何问题：
>
> ```
> Manager [] vam=new Manager[20];
> Employee [] vae=new Employee[50];
> vae=vam;
> ```
>
> 也就是数组里面，如果`T`是`F`的继承类，那么将一个`T[]`赋值`F[]`是完全没问题的

下面的部分内容来自CSDN:Java中<? extends T>和<? super T>的理解，下面会简称"CSDN文"

> 下面内容的参考文章：
>
> CSDN：关于 ? extends T 和 ? super T 的存在意义和使用
>
> CSDN：Java中<? extends T>和<? super T>的理解

先明确一个继承关系图，这样方便我们后面叙述



代码：

```
class Food{}
class Fruit extends Food{}
class Meat extends Food{}
class Pork extends Meat{}
class Beef extends Meat{}
class Apple extends Fruit{}
class Banana extends Fruit{}
class GreenApple extends Apple{}
class RedApple extends Apple{}
```

`<? extends T>`

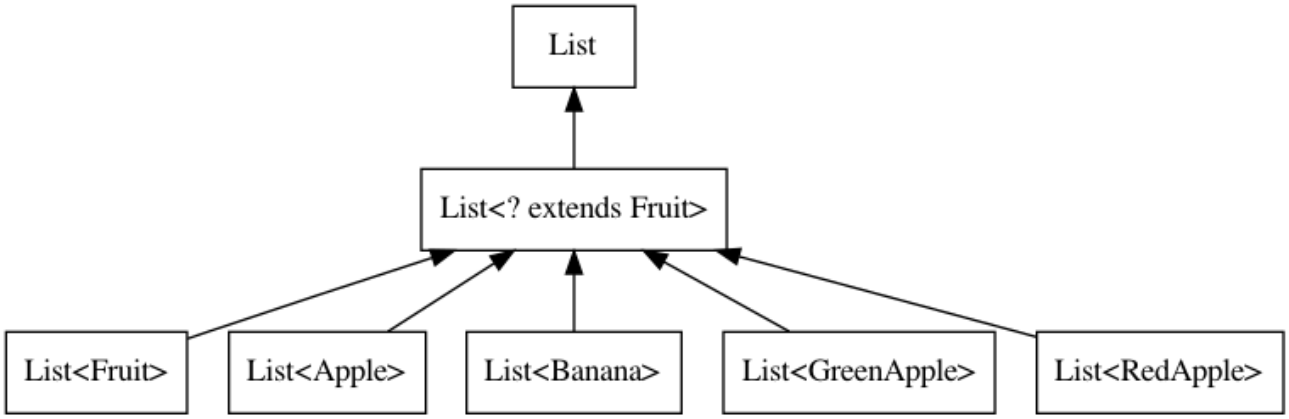
一种本质

这种通配符表示一种范型中特殊的继承祖先关系，比如：

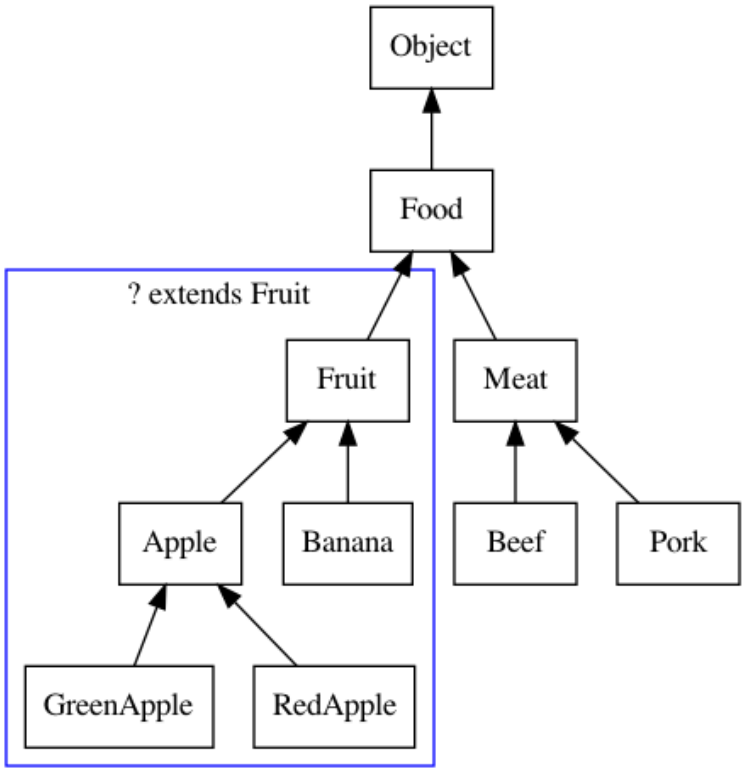
```
public class Wildcard {
    public static void main(String[] args) {
        List<? extends Fruit>list=getListFruit();
        // use list to do sth
        list=getListApple(); // #

    }
    static List<Apple>getListApple(){
        List<Apple>list=new LinkedList<>();
        return list;
```

```
    }
    static List<Fruit>getListFruit(){
        List<Fruit>list=new LinkedList<>();
        return list;
    }
}
```

现在的`#`那行代码不会报错了！这是因为，现在的`List<? extends Fruit>`和其他的`List<>`有了继承关系！如下图：



具体来说，在下图上，被蓝色框住的`T`就满足：“`List<T>`是`List<? extends T>`的继承类”：



类型拒绝

接着上面的代码，我们发现

```
list=getListApple();
list.add(new Banana());
list.add(new Fruit());
list.add(new Object());
```

后三个语句过不了编译！编译器输出`不兼容的类型：TestTemplate.Fruit无法转换为capture#1，共 ? extends TestTemplate.Fruit`

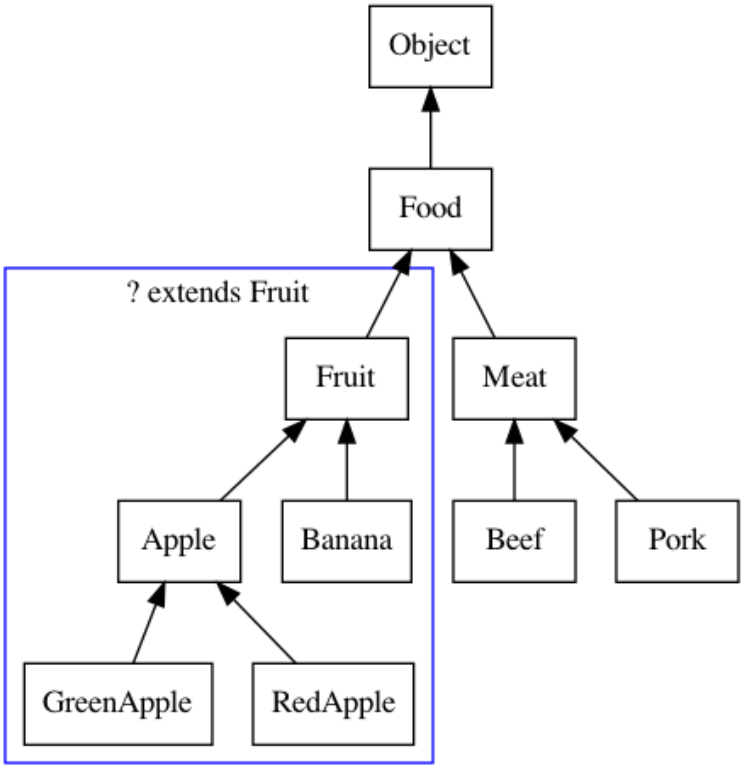
这是为什么呢？实际上`list`在运行时实际上是一个`List<T>`，其中`T`满足`extends Fruit`条件（即下图蓝色的部分中任何类），但是编译期编译器知道的只有：`list`类型是`List<? extends Fruit>`，那么`add(new Fruit())`这个语句就有问题了，编译器不知道运行时`list`里面的`T`是什么，到底能不能进行转换，自然就不让你过编译了。下面举例说明：

如果运行时`list`指向一个`List<Apple>`，然后你往里面加一个`Banana`，那么一定有问题，因为`Banana`和`Apple`之间不能互相转换

如果运行时`list`指向一个`List<Apple>`，然后你往里面加一个`Fruit`或者`Object`，此时也一定有问题，因为实际上`Fruit`和`Object`不能向下转型成`Apple`

如果运行时`list`指向一个`List<Apple>`，然后你往里面加一个`Apple`，这确实没问题，但是编译器编译期不知道“运行时`list`指向一个`List<Apple>`”！因此干脆一视同仁，简单化，让`List<? extends Fruit>`的`list`不能“接受”类型

这就是类型拒绝，你不能把任何类型的东西赋值给`list`里面`<? extends Fruit>`的变量



类型接受

> ~~这个是我自己创造的概念~~

一个不能加入的`List`有什么用呢？可以用来使用

```
for(int i=0;i<list.size();++i){
    Fruit fruit=list.get(i);
    // use fruit to do sth
}
```

这个语句可以过编译并且可以正常输出，也就是你可以用`Fruit`接受`list`里面`<? extends Fruit>`的变量，因为无论如何，`? extends Fruit`向上转型成`Fruit`都是合法的

这就是类型接受

`<? super T>`

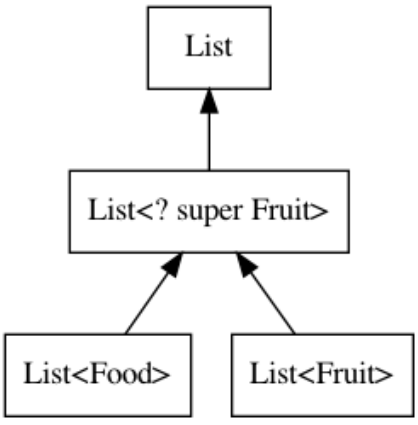
一种本质

和`<? extends T>`表示“这里接受一个继承自`T`或者实现了`T`的类”意义不同，`<? super T>`表示“这里接受一个基类有`T`的类”

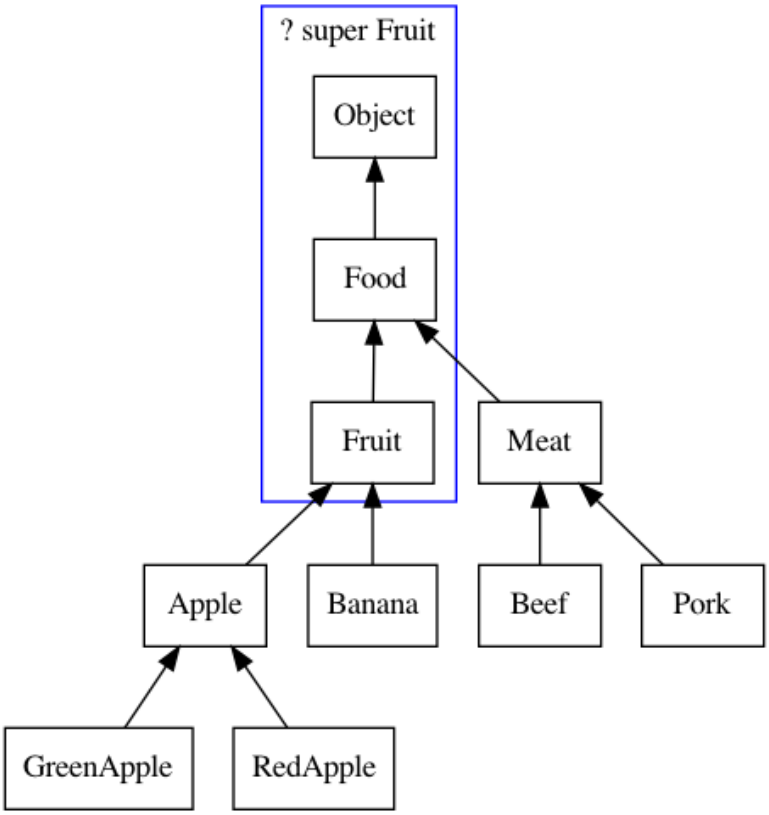
比如我们想实现一个“装`Fruti`或者有`Fruit`基类”的`List`，那么语句如下：

```
List<? super Fruit>list=getListFruit();
list=getListFood();
```

这时继承关系如下：



被蓝色框住的`T`就满足：“`List<T>`是`List<? super T>`的继承类”：



类型拒绝

同样，这里也有类型拒绝，不过这次不是给`? super Fruit`赋值时产生问题，而是在将`? super Fruit`赋值给其他元素的时候会有问题

简单说，我们仍然不知道`list`运行时指向的到底是什么，所以在“取元素”的时候会出问题，我们只能把取出的元素放在`Object`里

```
Fruit fruit=list.get(0);
Food food=list.get(0);
Object object=list.get(0); // #
```

上面三个句子只有`#`句可以过编译

类型接受

下面的语句是合法的

```
list.add(new Fruit());
```

因为无论如何`Fruit`向`? super Fruit`转型是合法的

PECS原则

最后看一下什么是PECS（Producer Extends Consumer Super）原则，已经很好理解了：

- 频繁往外读取内容的，适合用上界`extends`
- 经常往里插入的，适合用下界`super`