1. `Predicate<T>`

   断言函数，输入 T，返回 Boolean,常用方法为boolean test(T t),接口方法方法如下

   - `boolean test(T t);`

     根据给定的参数，带入断言中比较

     ```
     Predicate<Integer> predicate1 = i -> i < 10;
     // 等效写法
     //IntPredicate predicate1 = i -> i < 10;
     System.out.println(predicate1.test(11));
     // false
     ```

   - `default Predicate<T> and(Predicate other) { Objects.requireNonNull(other); return (t) -> test(t) && other.test(t); }`

     返回由两个断言构成的逻辑与断言，当执行该逻辑断言时，先执行第一个断言，如果第一个断言执行返回false，则第二个断言不会执行

     ```
     Predicate<Integer> predicate1 = i -> {
         System.out.println("执行了第一个断言");
         return i < 10;
     };

     Predicate<Integer> predicate2 = i -> {
         System.out.println("执行了第二个断言");
         return i < 8;
     };
     System.out.println(predicate1.and(predicate2).test(9));
     System.out.println("----------------------");
     System.out.println(predicate2.and(predicate1).test(9));

     // 执行了第一个断言
     // 执行了第二个断言
     // false
     // ----------------------
     // 执行了第二个断言
     // false
     ```

   - `default Predicate<T> negate() { return (t) -> !test(t); }`

     返回断言的逻辑非断言

     ```
     Predicate<Integer> predicate1 = i -> i < 10;
     //IntPredicate predicate1 = i -> i < 10;// 等效写法
     System.out.println(predicate1.negate().test(11));
     // true
     ```

   - `default Predicate<T> or(Predicate other) { Objects.requireNonNull(other); return (t) -> test(t) || other.test(t); }`

     返回由两个断言构成的逻辑或断言，当执行该逻辑断言时，先执行第一个断言，如果第一个断言执行返回true，则第二个断言不会执行

     ```
     Predicate<Integer> predicate1 = i -> {
         System.out.println("执行了第一个断言");
         return i < 10;
     };

     Predicate<Integer> predicate2 = i -> {
         System.out.println("执行了第二个断言");
         return i < 8;
     };
     System.out.println(predicate1.or(predicate2).test(9));
     System.out.println("----------------------");
     System.out.println(predicate2.or(predicate1).test(9));
     // 执行了第一个断言
     // true
     // ----------------------
     // 执行了第二个断言
     // 执行了第一个断言
     // true
     ```

- `static <T> Predicate<T> isEqual(Object targetRef) { return (null == targetRef) ? Objects::isNull : object -> targetRef.equals(object); }`

返回一个根据对象引用是否相等的测试断言

```
Object obj = new Object();
System.out.println(Predicate.isEqual(obj).test(new Object()));
System.out.println(Predicate.isEqual(obj).test(obj));
// false
// true
```

2. `Consumer<T>`

消费函数，输入 T，没有输出,常用方法为void accept(T t),接口方法方法如下

- `void accept(T t);`

  对于给定参数执行函数

  ```
  Consumer<String> consumer1 = System.out::println;
  consumer1.accept("hello world");
  // hello world
  ```

- `default Consumer<T> andThen(Consumer after) { Objects.requireNonNull(after); return (T t) -> { accept(t); after.accept(t); }; }`

  返回一个组合的Consumer，它按顺序执行函数，先执行第一个Consumer函数,然后执行后一个Consumer函数。如果执行任一函数抛出异常，它将被中继到组合函数的调用者。如果执行第一个Consumer函数会引发异常，将不会执行后一个Consumer函数。

  ```
  Consumer<String> consumer1 = t -> {
      int a = 1 / 0;
      System.out.println(t);
  };
  Consumer<String> consumer2 = System.out::println;
  // consumer1.andThen(consumer2).accept("hello world");
  consumer2.andThen(consumer1).accept("hello world");
  // hello world
  // Exception in thread "main" java.lang.ArithmeticException: / by zero
  ```

  、

3. `Function<T,R>`

单输入单输出,输入T,输出R,常用方法为R apply(T t),接口方法方法如下

- `R apply(T t);`

  对于给定的参数执行函数

  ```
  Function<Integer, Integer> function = i -> i + 1;
  System.out.println(function.apply(1));
  // 2
  ```

- `default <V> Function compose(Function before) { Objects.requireNonNull(before); return (V v) -> apply(before.apply(v)); }`

  返回一个组合函数,该函数按顺序将给定的参数应用到函数上,后一个函数的输入是前一个函数的输出.如果任一函数的评估抛出异常,则将其转发给组合函数的调用者

  ```
  Function<Integer, Integer> function1 = i -> i * 2;
  Function<Integer, Integer> function2 = i -> i + 1;
  System.out.println(function2.compose(function1).apply(1));
  System.out.println(function1.compose(function2).apply(1));
  // 3
  // 4
  ```

- `default <V> Function andThen(Function after) { Objects.requireNonNull(after); return (T t) -> after.apply(apply(t)); }`

  返回一个组合函数,该函数按从后到前的顺序将给定的参数应用到函数上,后一个函数的输入是前一个函数的输出.如果任一函数的评估抛出异常,则将其转发给组合函数的调用者

```
Function<Integer, Integer> function1 = i -> i * 2;
Function<Integer, Integer> function2 = i -> i + 1;
System.out.println(function2.andThen(function1).apply(1));
System.out.println(function1.andThen(function2).apply(1));
// 4
// 3
```

- `static <T> Function identity() { return t -> t; }`

  返回一个始终返回其输入参数的函数

```
System.out.println(Function.identity().apply(1));
// 1
```

4. `Supplier<T>`

   无输入，输出 T，典型的生产者模型,接口方法如下

   - `T get();`

     获取函数结果

```
Supplier<String> stringSupplier = () -> "hello world";
System.out.println(stringSupplier.get());
// hello world
```

5. `UnaryOperator<T>`

   输出、输入均为 T,继承了Function接口

6. `BiFunction`

   表示接受两个参数并生成结果的函数,常用方法为R apply(T t, U u),接口方法如下

   - `R apply(T t, U u);`

     将此函数应用于给定的两个参数

```
BiFunction<String, String, Integer> biFunction = (arg1, arg2) -> Integer.parseInt(arg1) + Integer.parseInt(arg2);
System.out.println(biFunction.apply("1", "1"));
// 2
```

   - `default <V> BiFunction andThen(Function after) { Objects.requireNonNull(after); return (T t, U u) -> after.apply(apply(t, u)); }`

     返回一个组合函数,该函数首先将此函数应用于其输入，然后将后一个函数应用于结果.如果任一函数的抛出异常,则将其转发给组合函数的调用者.

```
BiFunction<String, String, Integer> biFunction = (arg1, arg2) -> Integer.parseInt(arg1) + Integer.parseInt(arg2);
Function<Integer, Integer> function = arg -> arg * 2;
System.out.println(biFunction.andThen(function).apply("1", "1"));
// 4
```

7. `BinaryOperator<T>`

   继承BiFunction接口,表示对两个相同类型的操作数的操作，产生与操作数相同类型的结果.对于操作数和结果都是相同类型的情况，这是BiFunction的特例.特有接口如下

   - `public static <T> BinaryOperator minBy(Comparator comparator) { Objects.requireNonNull(comparator); return (a, b) -> comparator.compare(a, b) <= 0 ? a : b; }`

     返回BinaryOperator，它根据指定的Comparator返回两个元素中的较小者.

   - `public static <T> BinaryOperator maxBy(Comparator comparator) { Objects.requireNonNull(comparator); return (a, b) -> comparator.compare(a, b) >= 0 ? a : b; }`

     返回BinaryOperator，它根据指定的Comparator返回两个元素中的较大者.