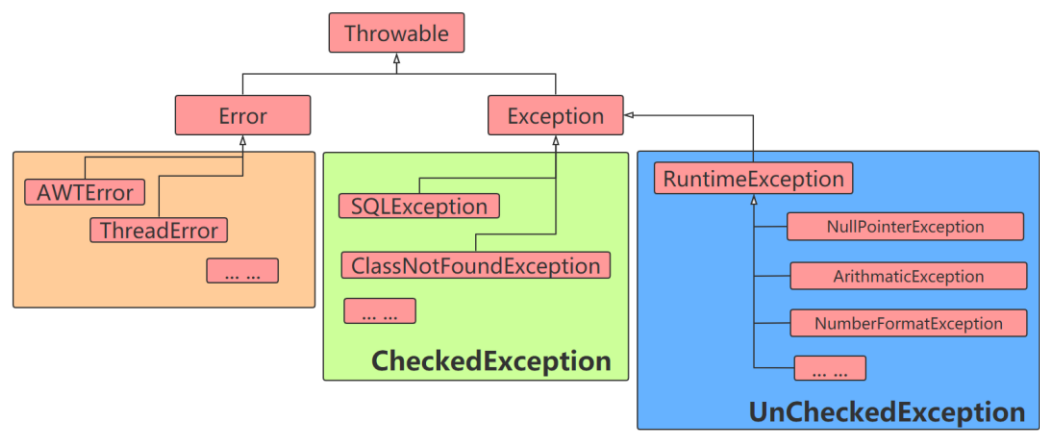


01 异常机制



大纲	知识点
异常机制	异常机制的本质
	异常的概念
	异常的分类
异常的处理	try-catch-finally
	throws 声明异常
	try-with-resource 新特性
	自定义异常
百度处理异常	处理异常的步骤
	百度：超级搜索
debug 调试模式	可视化调试



导引问题

工作中，程序遇到的情况不可能完美。比如：程序要打开某个文件，这个文件可能不存在或者文件格式不对；程序在运行着，但是内存或硬盘可能满了等等。

软件程序在运行过程中，非常可能遇到刚刚提到的这些问题，我们称之为异常，英文是：Exception，意思是例外。遇到这些例外情况，或者叫异常，我们怎么让写的程序做出合理的处理，安全的退出，而不至于程序崩溃呢？我们本章就要讲解这些问题。

如果我们要拷贝一个文件，在没有异常机制的情况下，我们需要考虑各种异常情况，伪代码如下：

【示例】伪代码：使用 if 处理程序中可能出现的各种情况

d:/a.txt复制到e:/a.txt(伪代码)

```
if("d:/a.txt"这个文件存在){
    if(e盘的空间大于a.txt文件长度){
        if(文件复制一半IO流断掉){
            停止copy, 输出: IO流出问题!
        }else{
            copyFile("d:/a.txt", "e:/a.txt");
        }
    }else{
        System.out.println("e盘空间不够存放a.txt! ");
    }
}else{
    System.out.println("a.txt不存在! ");
}
```

这种方式，有两个坏处：

1. 逻辑代码和错误处理代码放一起！
2. 程序员本身需要考虑的例外情况较复杂，对程序员本身要求较高！

如上情况，如果是用 Java 的异常机制来处理，对比如下：

d:/a.txt复制到e:/a.txt(伪代码)

```
if("d:/a.txt"这个文件存在){
    if(e盘的空间大于a.txt文件长度){
        if(文件复制一半IO流断掉){
            停止copy, 输出: IO流出问题!
        }else{
            copyFile("d:/a.txt", "e:/a.txt");
        }
    }else{
        System.out.println("e盘空间不够存放a.txt! ");
    }
}else{
    System.out.println("a.txt不存在! ");
}
```

使用异常机制处理

```
try {
    copyFile("d:/a.txt", "e:/a.txt");
} catch (Exception e) {
    e.printStackTrace();
}
```

异常机制本质

当程序出现异常，程序安全的退出、处理完后继续执行的机制

异常 (Exception) 的概念

异常指程序运行过程中出现的非正常现象，例如除数为零、需要处理的文件不存在、数组下标越界等。

在 Java 的异常处理机制中，引进了很多用来描述和处理异常的类，称为异常类。异常类定义中包含了该类异常的信息和对异常进行处理的方法。

我们开始看我们的第一个异常对象，并分析一下异常机制是如何工作的。

【示例】异常的分析

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println("111");  
        int a = 1/0;  
        System.out.println("222");  
    }  
}
```

执行结果如图所示：

```
111  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at com.bjsxt.Test.main(Test.java:6)  
  
Process finished with exit code 1
```

根据结果，我们可以看到执行“1/0”时发生了异常，程序终止了，没有执行后面的打印“222”的动作。

如果我们使用 try-catch 来处理，程序遇到异常可以正常的处理，处理完成后，程序继续往下执行：

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println("111");  
        try {  
            int a = 1/0;  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
    }  
    System.out.println("222");  
}  
}
```

执行结果如下：

```
111  
java.lang.ArithmeticException: / by zero  
    at com.bjsxt.Test.main(Test.java:7)  
222  
  
Process finished with exit code 0
```

程序在执行“1/0”仍然遇到异常，然后进行 try-catch 处理。处理完毕后，程序继续往下执行，打印了“222”内容。

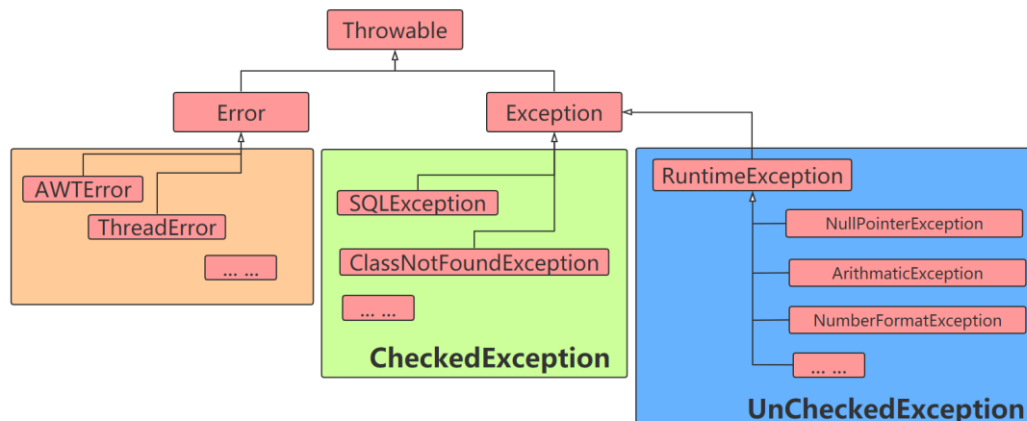
Java 是采用面向对象的方式来处理异常的。处理过程：

- **抛出异常：**在执行一个方法时，如果发生异常，则这个方法生成代表该异常的一个对象，停止当前执行路径，并把异常对象提交给 JRE。
- **捕获异常：**JRE 得到该异常后，寻找相应的代码来处理该异常。JRE 在方法的调用栈中查找，从生成异常的方法开始回溯，直到找到相应的异常处理代码为止。

异常分类

Java 中定义了很多异常类，这些类对应了各种各样可能出现的异常事件，所有异常对象都是派生于 Throwable 类的一个实例。如果内置的异常类不能够满足需要，还可以创建自己的异常类。

Java 对异常进行了分类，不同类型的异常分别用不同的 Java 类表示，所有异常的根类为 **java.lang.Throwable**，Throwable 下面又派生了两个子类：Error 和 Exception。Java 异常类的层次结构如图所示：



Error

Error 是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM (Java 虚拟机) 出现的问题。例如，Java 虚拟机运行错误 (Virtual MachineError)，当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java 虚拟机 (JVM) 一般会选择线程终止。

Error 表明系统 JVM 已经处于不可恢复的崩溃状态中。

`java.lang`

类 Error

`java.lang.Object`

└ `java.lang.Throwable`

└ `java.lang.Error`

所有已实现的接口：

[`Serializable`](#)

直接已知子类：

[`AnnotationFormatError`](#), [`AssertionError`](#), [`AWTError`](#), [`CoderMalfunctionError`](#), [`FactoryConfigurationError`](#), [`FactoryConfigurationError`](#), [`IOError`](#), [`LinkageError`](#), [`ServiceConfigurationError`](#), [`ThreadDeath`](#), [`TransformerFactoryConfigurationError`](#), [`VirtualMachineError`](#)

`java.lang` 包中 Error 的类

Error 与 Exception 的区别

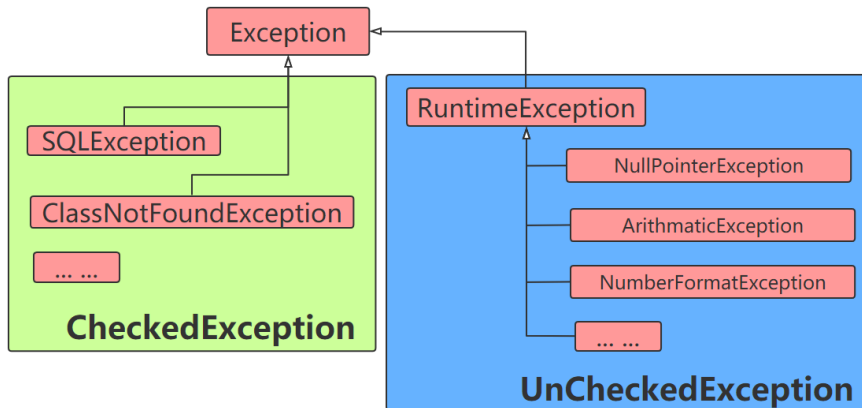
- ❑ 我开着车走在路上，一头猪冲在路中间，我刹车。这叫一个异常。
- ❑ 我开着车在路上，发动机坏了，我停车，这叫错误。系统处于不可恢复的崩溃状态。发动机什么时候坏？我们普通司机能管吗？不能。发动机什么时候坏是汽车厂发动机制造商的事。

Exception

Exception 是程序本身能够处理的异常。

Exception 类是所有异常类的父类，其子类对应了各种各样可能出现的异常事件。通常 Java 的异常可分为：

1. RuntimeException 运行时异常
2. CheckedException 已检查异常



RuntimeException 运行时异常

派生于 RuntimeException 的异常，如被 0 除、数组下标越界、空指针等，其产生比较频繁，处理麻烦，如果显式的声明或捕获将会对程序可读性和运行效率影响很大。因此由系统自动检测并将它们交给缺省的异常处理程序。

编译器不处理 RuntimeException，程序员需要增加“逻辑处理来避免这些异常”。

【示例】ArithmeticException 异常：试图除以 0

```
public class Test3 {  
    public static void main(String[] args) {  
        int b=0;  
        System.out.println(1/b);  
    }  
}
```

执行结果如图所示：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Test3.main(Test3.java:4)
```

解决如上异常需要修改代码：

```
public class Test3 {  
    public static void main(String[] args) {
```

```

        int b=0;
        if(b!=0){
            System.out.println(1/b);
        }
    }
}

```

【示例】NullPointerException 异常

```

public class Test4 {
    public static void main(String[] args) {
        String str=null;
        System.out.println(str.charAt(0));
    }
}

```

执行结果如图所示：

```

Exception in thread "main" java.lang.NullPointerException
    at Test4.main(Test4.java:4)

```

解决空指针异常，通常是增加非空判断：

```

public class Test4 {
    public static void main(String[] args) {
        String str=null;
        if(str!=null){
            System.out.println(str.charAt(0));
        }
    }
}

```

【示例】ClassCastException 异常

```

class Animal{

}

class Dog extends Animal{

}

class Cat extends Animal{

}

public class Test5 {

```

```

    public static void main(String[] args) {
        Animal a=new Dog();
        Cat c=(Cat)a;
    }
}

```

执行结果如图所示：

```

Exception in thread "main" java.lang.ClassCastException: Dog cannot be cast to Cat
at Test5.main(Test5.java:4)

```

解决 ClassCastException 的典型方式：

```

public class Test5 {
    public static void main(String[] args) {
        Animal a = new Dog();
        if (a instanceof Cat) {
            Cat c = (Cat) a;
        }
    }
}

```

【示例】ArrayIndexOutOfBoundsException 异常

```

public class Test6 {
    public static void main(String[] args) {
        int[] arr = new int[5];
        System.out.println(arr[5]);
    }
}

```

执行结果如图所示：

```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
at Test6.main(Test6.java:4)

```

解决数组索引越界异常的方式，增加关于边界的判断：

```

public class Test6 {
    public static void main(String[] args) {
        int[] arr = new int[5];
        int a = 5;
        if (a < arr.length) {
            System.out.println(arr[a]);
        }
    }
}

```



```
}  
}
```

【示例】NumberFormatException 异常

```
public class Test7 {  
    public static void main(String[] args) {  
        String str = "1234abcf";  
        System.out.println(Integer.parseInt(str));  
    }  
}
```

执行结果如图所示：

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "1234abcf"  
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
    at java.lang.Integer.parseInt(Integer.java:580)  
    at java.lang.Integer.parseInt(Integer.java:615)  
    at Test7.main(Test7.java:4)
```

数字格式化异常的解决，可以引入正则表达式判断是否为数字：

```
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
  
public class Test7 {  
    public static void main(String[] args) {  
        String str = "1234abcf";  
        Pattern p = Pattern.compile("^\\d+$");  
        Matcher m = p.matcher(str);  
        if (m.matches()) { // 如果 str 匹配代表数字的正则表达式,才会转换  
            System.out.println(Integer.parseInt(str));  
        }  
    }  
}
```

CheckedException 已检查异常



CheckedException 异常在编译时就必须处理，否则无法通过编译。如图所示。

```
File f = new File(pathname: "d:/a.txt");  
f.createNewFile();
```

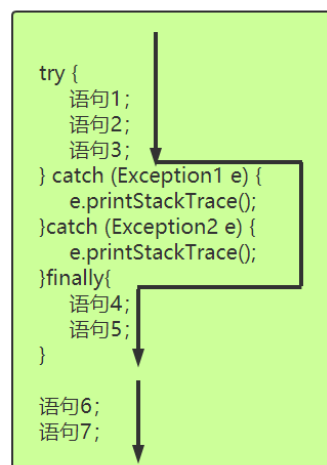
! Add exception to method signature
! Surround with try/catch

CheckedException 异常的处理方式有两种：

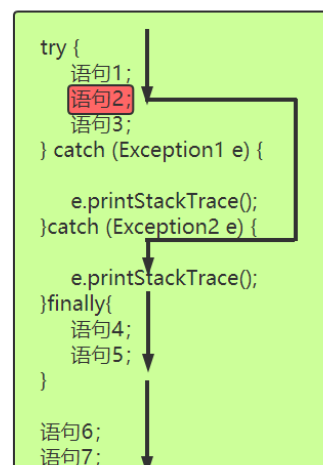
1. 使用 “try/catch” 捕获异常
2. 使用 “throws” 声明异常。

异常的处理方式之一：捕获异常

若无异常，代码执行顺序



若语句2有异常，代码执行顺序



□ try:

try 语句指定了一段代码，该段代码就是异常捕获并处理的范围。在执行过程中，当任意一条语句产生异常时，就会跳过该条语句中后面的代码。代码中可能会产生并抛出一种或

几种类型的异常对象，它后面的 catch 语句要分别对这些异常做相应的处理。

一个 try 语句必须带有至少一个 catch 语句块或一个 finally 语句块。

注意事项

- ❑ 当异常处理的代码执行结束以后，不会回到 try 语句去执行尚未执行的代码。

❑ catch:

- 每个 try 语句块可以伴随一个或多个 catch 语句，用于处理可能产生的不同类型的异常对象。
- catch 捕获异常时的捕获顺序：
 - ◆ 如果异常类之间有继承关系，先捕获子类异常再捕获父类异常。

❑ finally:

- 不管是否发生了异常，都必须执行。
- 通常在 finally 中关闭已打开的资源，比如：关闭文件流、释放数据库连接等。

try-catch-finally 语句块的执行过程详细分析：

程序首先执行可能发生异常的 try 语句块。如果 try 语句没有出现异常则执行完后跳至 finally 语句块执行；如果 try 语句出现异常，则中断执行并根据发生的异常类型跳至相应的 catch 语句块执行处理。catch 语句块可以有多个，分别捕获不同类型的异常。catch 语句块执行完后程序会继续执行 finally 语句块。finally 语句是可选的，如果有的话，则不管是否发生异常，finally 语句都会被执行。

【示例】异常处理的典型代码(捕获异常)

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class Test8 {
    public static void main(String[] args) {
        FileReader reader = null;
        try {
            reader = new FileReader("d:/a.txt");
            char c = (char) reader.read();
            char c2 = (char) reader.read();
            System.out.println(" " + c + c2);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    } finally {
        try {
            if (reader != null) {
                reader.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}

```

常用开发环境中，自动增加 try-catch 代码块的快捷键：

1. 将需要处理异常的代码选中。
2. IDEA 中，使用：ctrl+alt+t
3. eclipse 中，使用：ctrl+shift+z

异常的处理方式之二：声明异常（throws 子句）



1. CheckedException 产生时，不一定立刻处理它，可以把异常 throws，由调用者处理。
2. 一个方法抛出多个已检查异常，就必须在方法的首部列出所有的异常。

【示例】异常处理的典型代码（声明异常抛出 throws）

```

package com.bjsxt;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

```

```

public class Test9 {
    public static void main(String[] args) {
        try {
            readFile("joke.txt");
        } catch (FileNotFoundException e) {
            System.out.println("所需文件不存在! ");
        } catch (IOException e) {
            System.out.println("文件读写错误! ");
        }
    }

    public static void readFile(String fileName) throws
FileNotFoundException,
        IOException {
        FileReader in = new FileReader(fileName);
        int tem = 0;
        try {
            tem = in.read();
            while (tem != -1) {
                System.out.print((char) tem);
                tem = in.read();
            }
        } finally {
            if(in!=null) {
                in.close();
            }
        }
    }
}

```

注意事项

- ❑ 方法重写中声明异常原则：子类重写父类方法时，如果父类方法有声明异常，那么子类声明的异常范围不能超过父类声明的范围。

try-with-resource 自动关闭 AutoClosable 接口的资源

JAVA 中，JVM 的垃圾回收机制可以对内部资源实现自动回收，给开发者带来了极大的便利。但是 JVM 对外部资源(调用了底层操作系统的资源)的引用却无法自动回收，例如数据库连接，网络连接以及输入输出 IO 流等。这些连接就需要我们手动去关闭，不然会导致外部资源泄露，连接池溢出以及文件被异常占用等。

JDK7 之后，新增了“try-with-resource”。它可以自动关闭实现了 AutoClosable 接口的类，实现类需要实现 close()方法。“try-with-resources 声明”，将 try-catch-finally 简化为 try-catch，这其实是一种语法糖，在编译时仍然会进行转化为 try-catch-finally 语句。

```
package com.bjsxt;

import java.io.FileReader;
public class Test8 {
    public static void main(String[] args) {
        try(FileReader reader = new FileReader("d:/a.txt");) {
            char c = (char) reader.read();
            char c2 = (char) reader.read();
            System.out.println("" + c + c2);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

自定义异常

- ❑ 在程序中，可能会遇到 JDK 提供的任何标准异常类都无法充分描述清楚我们想要表达的问题，这种情况下可以创建自己的异常类，即自定义异常类。
- ❑ 自定义异常类只需从 Exception 类或者它的子类派生一个子类即可。
- ❑ 自定义异常类如果继承 Exception 类，则为 CheckedException 异常，必须对其进行处理；如果不想处理，可以让自定义异常类继承运行时异常 RuntimeException 类。

- 习惯上, 自定义异常类应该包含 2 个构造器: 一个是默认的构造器, 另一个是带有详细信息的构造器。

【示例】自定义异常类

```
/**IllegalAgeException: 非法年龄异常, 继承 Exception 类*/
public class IllegalAgeException extends Exception {
    //默认构造器
    public IllegalAgeException() {

    }
    //带有详细信息的构造器, 信息存储在 message 中
    public IllegalAgeException(String message) {
        super(message);
    }
}
```

【示例】自定义异常类的使用

```
class Person {
    private String name;
    private int age;

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) throws IllegalAgeException {
        if (age < 0) {
            throw new IllegalAgeException("人的年龄不应该为负数");
        }
        this.age = age;
    }

    public String toString() {
        return "name is " + name + " and age is " + age;
    }
}

public class TestMyException {
    public static void main(String[] args) {
```

```

Person p = new Person();
try {
    p.setName("Lincoln");
    p.setAge(-1);
} catch (IllegalAgeException e) {
    e.printStackTrace();
}
System.out.println(p);
}

```

执行结果如图所示：

```

IllegalAgeException: 人的年龄不应该为负数
    at Person.setAge(TestMyException.java:24)
    at TestMyException.main(TestMyException.java:38)

```

使用异常机制的建议

- ❑ 要避免使用异常处理代替错误处理，这样会降低程序的清晰性，并且效率低下。
- ❑ 处理异常不可以代替简单测试---只在异常情况下使用异常机制。
- ❑ 不要进行小粒度的异常处理---应该将整个任务包装在一个 try 语句块中。
- ❑ 异常往往在高层处理(先了解！后面做项目会说！)。

如何利用百度解决异常问题



内事不决问老婆，外事不决问百度

正常学习和开发中，我们经常会遇到各种异常。遇到异常时，需要遵循下面四步来解决：

1. 细心查看异常信息，确定异常种类和相关 Java 代码行号

2. 确定上下文相关的一些关键词信息（疑难问题，需要）。
拷贝异常信息到百度，查看相关帖子，寻找解决思路；
3. 前两步无法搞定，再问同学/老师或同事；
4. 前三步无法搞定，请示领导。

很多同学碰到异常一下就慌了，立刻开始请教别人搬救兵，殊不知这样做有两大坏处。第一、太不尊重别人，把别人当苦力。第二、失去提高自我的机会，自己解决一个异常，就意味着有能力解决一类异常。解决一类异常能大大提高自身能力。

不要怕花时间在解决问题上，不要觉得解决问题是耽误时间。解决问题的过程中，本身你也在思考。

• 百度超级搜索：

百度/Google 搜索用好的关键是：关键词的确认，正确的提问。

1. 寻找问题本身的关键词（名词）
2. 寻找问题上下文的关键词（名词）
3. 尽量细致的描述问题，开始搜索
4. 如果没找到，慢慢减少关键词，扩大搜索范围。

IDEA 调试 debug



调试的核心是断点。程序执行到断点时，暂时挂起，停止执行。就像看视频按下停止一样，我们可以详细的观看停止处的每一个细节。

断点 breakpoint

程序运行到此处，暂时挂起，停止执行。我们可以详细在此时观察程序的运行情况，方

便做出进一步的判断。

1. 设置断点:


(1) 在行号后面单击即可增加断点

```
8      private int num = 10;    num: 10
9
10     public void run(int a){    a: 3
11         System.out.println("Test05.run");
12         for(int i=0;i<a;i++){
13             num += a;
14             go();
15         }
16     }
17
18     public void go() {
19         System.out.println("num="+num);
20         System.out.println("num*10="+num*10);
21     }
22
```

(2) 在断点上再单击即可取消断点

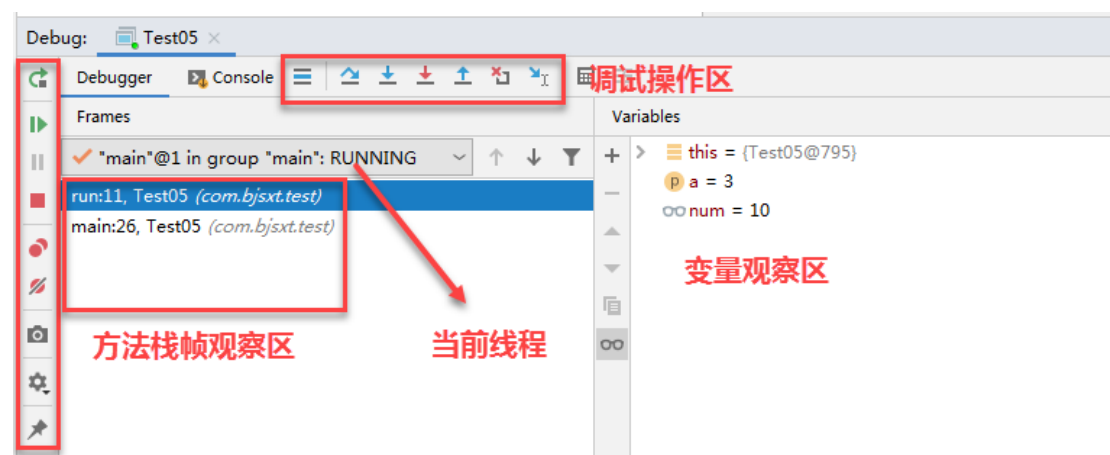
进入调试视图

我们通过如下三种方式都可以进入调试视图:

(1) 单击工具栏上的按钮: 

(2) 右键单击编辑区, 点击: debug

进入调试视图后, 布局如下:



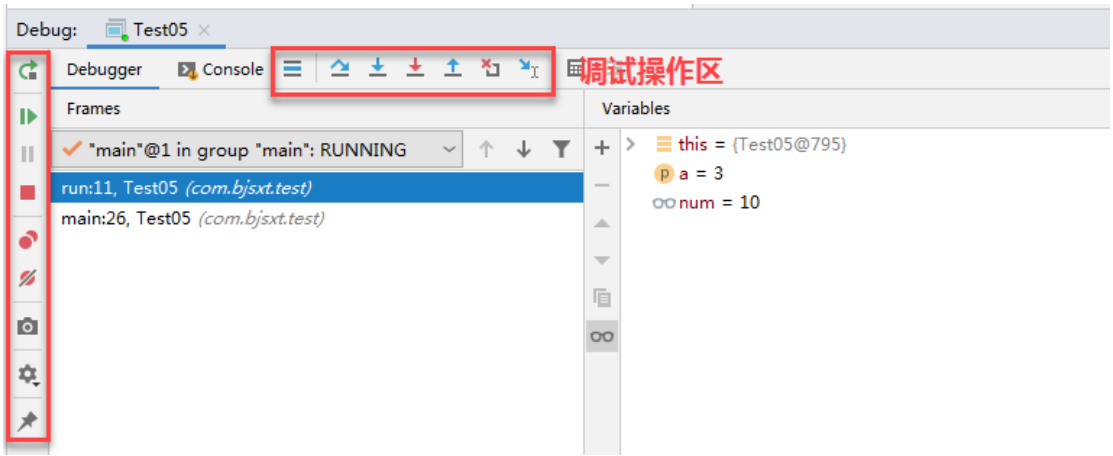
左侧为“浏览帧”：

调试器列出断点处，当前线程正在运行的方法，每个方法对应一个“栈帧”。最上面的是当前断点所处的方法。

变量值观察区：

调试器列出了断点处所在方法相关的变量值。我们可以通过它，查看变量的值的变化。

调试操作区



我们通过上图中的按钮进行调试操作，它们的含义如下：

中文名称	英文名称	图标	说明
单步调试： 遇到方法跳过	step over		若当前执行的是一个方法，则会把这个方法当做整体一步执行完。不会进入这个方法内部
单步调试： 遇到函数进入	step into		若当前执行的是一个自定义方法，则会进入方法内部。JDK 内置方法不会进去

强制进入	force step into		可以跳进任何方法，包含 JDK 内置方法
跳出函数	step out		当单步执行到子方法内时，用 step out 就可以执行完子方法余下部分，并返回到上一层方法
删除栈帧	drop frame		删除当前栈帧。跳回到上一个栈帧
执行的光标处	run to cursor		一直执行，到光标处停止，用在循环内部时，点击一次就执行一个循环
重新执行程序	rerun		重新执行所有程序
继续执行	resume		继续执行到下一个断点或者程序结束
停止程序	stop		
查看所有断点信息	view breakpoints		