第 5 章

调试与运行

用户在开发过程中会遇到各种问题，其中有些问题比较容易快速定位和修复，但更多的问题却无法通过观察或经验快速分析出错误原因，因此学会对项目代码进行调试显得尤为重要。

通常情况下项目是作为整体来运行的，而调试过程则主要是基于代码片断进行局部分析。在调试目标由整体下降到了局部级别后，使用最多的就是单元测试，而由于测试方式的不同又可以分为运行模式(Run)与调试模式(Debug)两种。

在 IntelliJ IDEA 中运行与调试都是基于配置进行的，每种配置都对应于某个特定调试目标的属性集合，用户可以创建不同类型的配置以运行对应的程序并进行调试。

5.1 测试目录

IntelliJ IDEA 项目结构中包含指定的测试目录，开发者通常会将自定义的测试用例放在测试目录下进行统一管理，如图 5.1 所示。

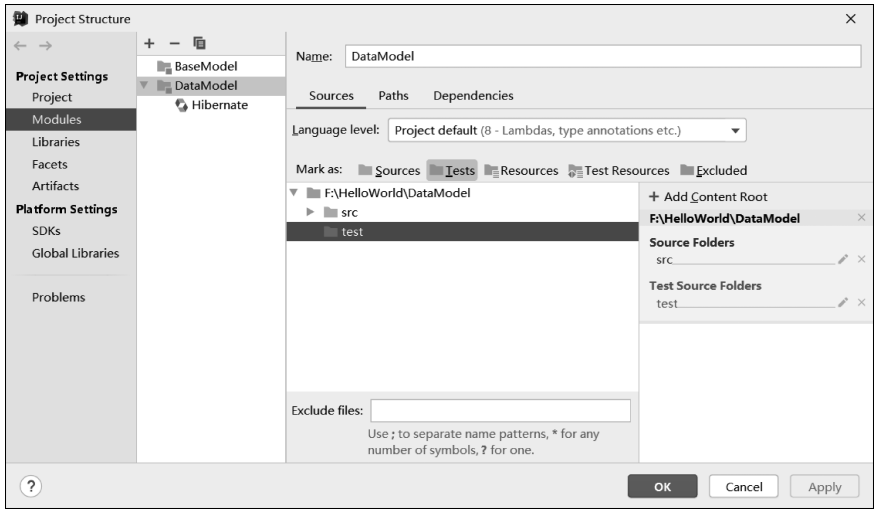

图 5.1 指定测试目录

5.2　运行/调试配置

运行/调试的基础是配置，IntelliJ IDEA 提供了运行/调试配置工具来创建与测试相关的配置。执行菜单 Run→Debug Configuration 命令打开运行/调试配置窗口，如图 5.2 所示。

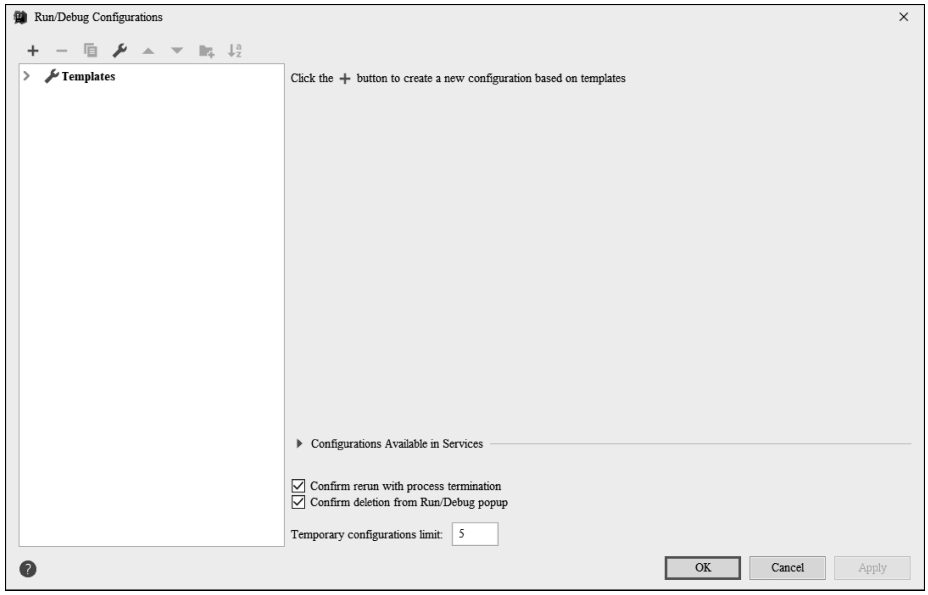

图 5.2　运行/调试配置窗口

除了上述方式外，单击工具栏的运行/调试配置列表也可以打开配置对话框。在未添加任何配置时运行/调试配置列表默认显示 Add Configuration，如果配置存在，则可下拉选择 Edit Configuration 菜单打开运行/调试配置窗口，如图 5.3 所示。

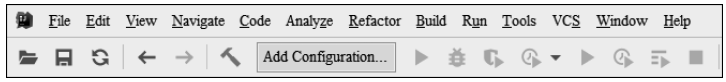

图 5.3　工具栏配置

在运行/调试配置窗口中单击 + 按钮或使用快捷键 Alt+Insert 展开配置类型列表，如图 5.4 所示。

选择需要的配置类型。如果需要创建 JUnit 单元测试，就选择 JUnit 配置类型。如果需要创建可运行程序，就选择 Application 配置类型，如图 5.5 所示。

其中，Name 用于指定测试配置的名称，默认为 Unnamed，配置完成并保存后此名称会显示在工具栏中的运行/调试配置列表中。

Allow parallel run 选项用于指定是否允许多实例运行，此选项通常在需要开启多个实例同时测试的时候使用，默认使用单实例方式测试。

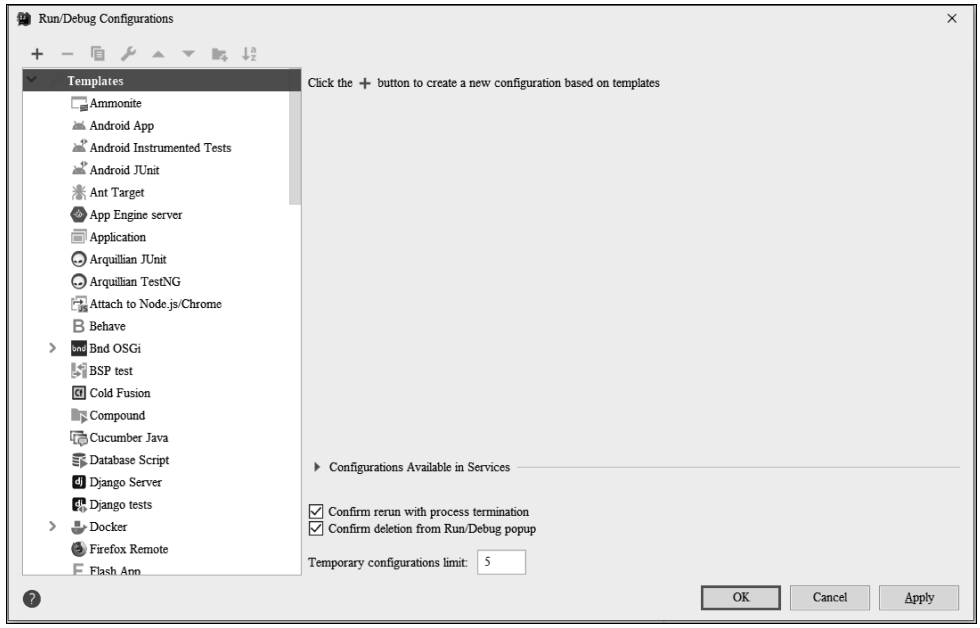

图 5.4　选择配置类型

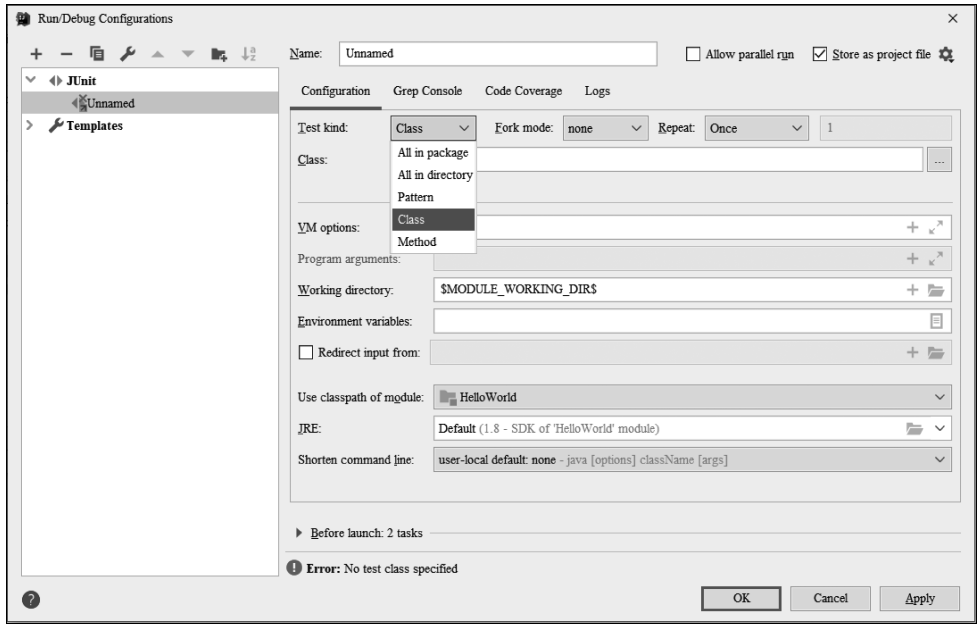

图 5.5　自定义配置(一)

Testkind 指定测试实例的来源类型，其中 All in packages 指定固定包下的所有可执行测试用例，使用此选项时需要指定整个项目级别或模块级别的完整包路径，如图 5.6 所示。

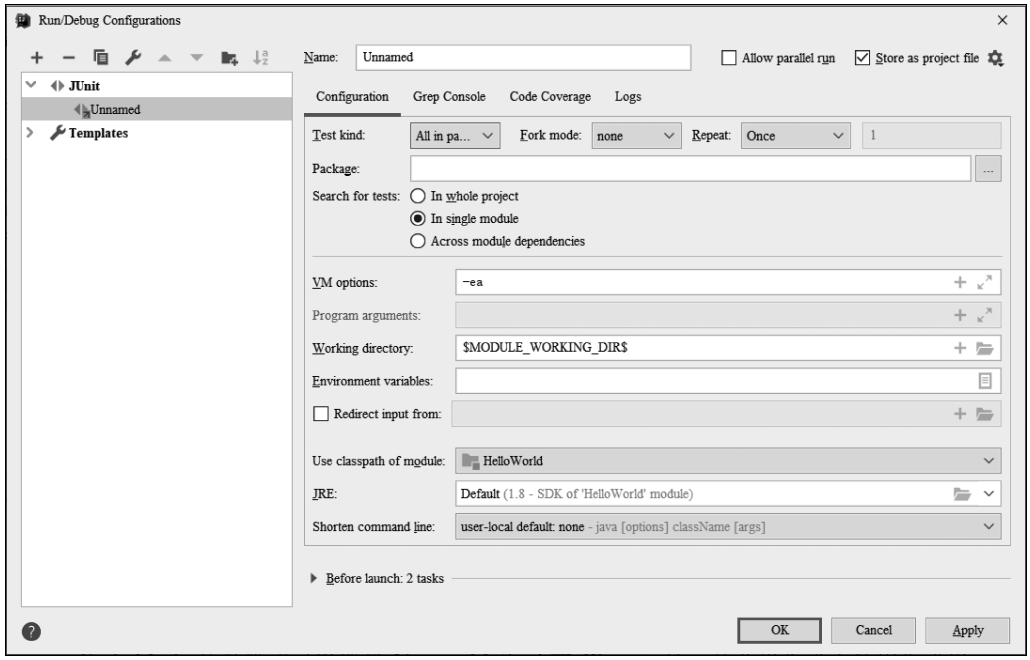

图 5.6　自定义配置(二)

Package 文本框内用于指定对应访问的目录或通过右侧快捷访问窗口进行选择。

Fork Mode 分叉模式可以选择 None、Method 或 Class 3 种，其含义为“不使用”“以方法级别”或“以类级别”启动线程来运行测试用例。

Repeat 用于指定重复执行的次数，其值为 Once(一次)、N Times(多次)、Until Failure(直至失败)和 Until Stopped(直至停止)4 种。如果选择 N Times 则需要在右侧输入具体的执行次数，默认为 1 次。

Code Coverage 选项卡用于代码覆盖率配置，其中指定了多个代码覆盖率检测插件，默认为 IntelliJ IDEA 自带插件，通过这些插件可以进行代码的行覆盖率、方法覆盖率、类覆盖率、元素覆盖率等多种统计，如图 5.7 所示。

当测试程序运行时可以选择带有代码覆盖率检测的运行方式，如图 5.8 所示。

当程序运行完成后会生成统计数据，如图 5.9 所示。

用户还可以将生成的统计测试结果导出到 HTML 中生成统计测试报告，如图 5.10 所示。

Logs 选项卡指定了程序运行/调试生成的日志输出等配置，如输出到控制台或保存到指定输出文件中，如图 5.11 所示。

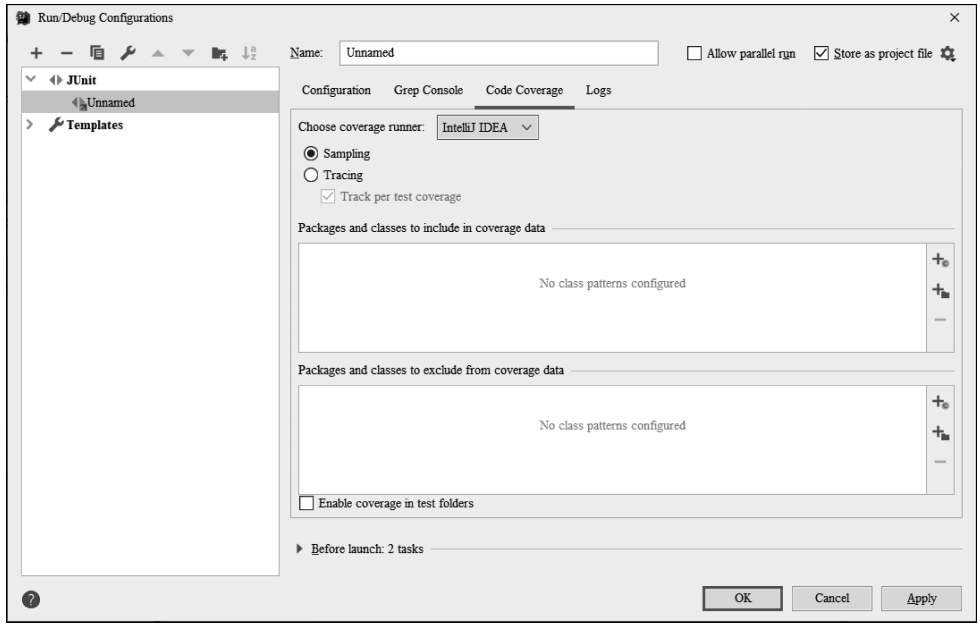

图 5.7　代码覆盖率选项

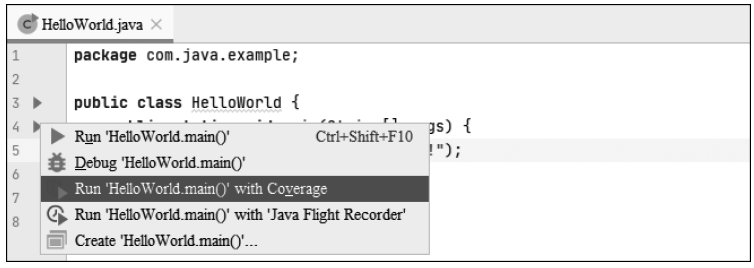

图 5.8　执行代码覆盖率方法

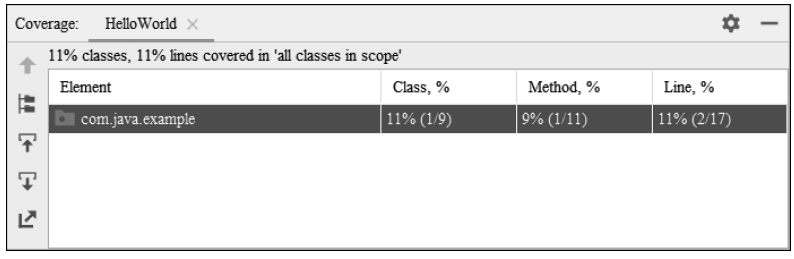

图 5.9　代码覆盖率统计

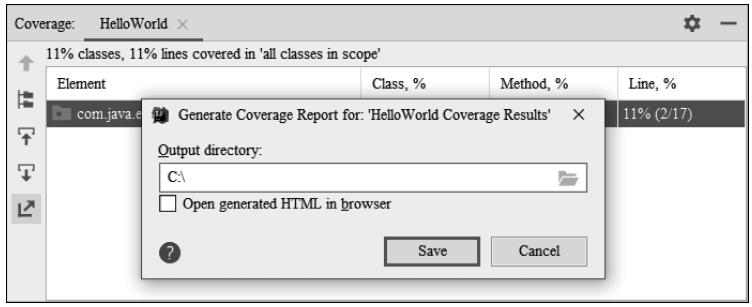

图 5.10　导出代码覆盖率统计报告

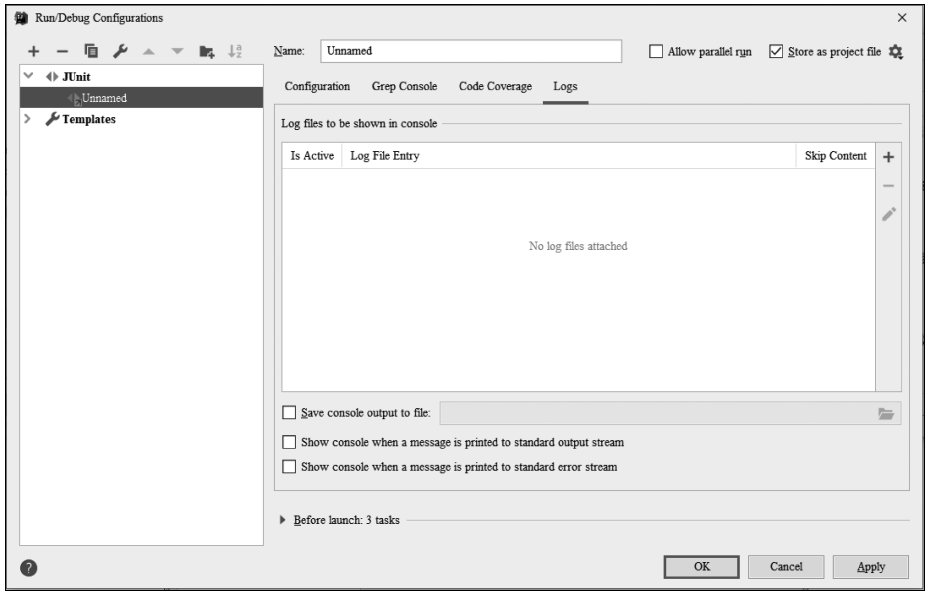

图 5.11　配置日志选项

5.3　Debug 调试

Debug 模式用来追踪程序运行并进行细粒度分析，最终定位到异常发生的位置。在调试过程中可以实时观察到各项参数值的变化，还可以根据需要进行参数的动态调整与计算。

结合其他框架进行调试时，Debug 模式还可以深入到框架内部追踪执行过程并定位问题。例如作者在基于 Spring 框架访问 Rest 服务的过程中，外部服务响应只返回了 gzip 压缩标识而并未对内容进行压缩，而 Spring RestTemplate 在识别到 gzip 压缩标识时会自动解压返回的报文内容，因此产生了难以定位的异常问题。如果没有 Debug 强大的调试功能，则这种问题通常很难被发现，尤其是在多级服务分离的环境下。

Debug 调试模式离不开断点，断点提供了在程序执行到不同位置时的阻塞状态，用户可以基于断点查看程序运行的状态并且有步骤地进行调试，并最终确定调试结果与预期是否一致。

IntelliJ IDEA 为断点的调试提供了强大的功能支持，如可自定义断点的属性、表达式的实时计算、变量值的动态更改等。

5.3.1 Debug 窗口布局

在进行单元测试时，Debug 模式可由工具栏上的 图标按钮启动并触发，也可以在快速启动或右击菜单中选择 Debug 命令运行，如图 5.12、图 5.13、图 5.14 所示。

图 5.12 Debug 模式

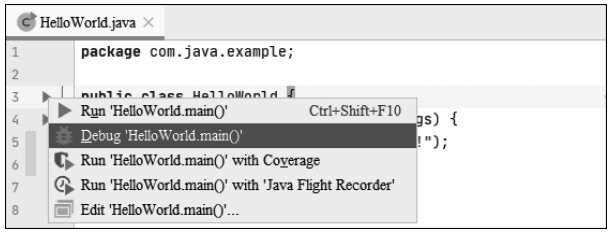

图 5.13 快速启动执行 Debug 命令

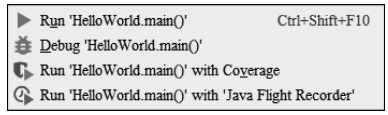

图 5.14 右击执行 Debug 命令

当程序以 Debug 方式运行后系统会展开 Debug 调试窗口来帮助开发者进行调试，如图 5.15 所示，图中展示了进行 Debug 调试时所涉及的几个关键区域。

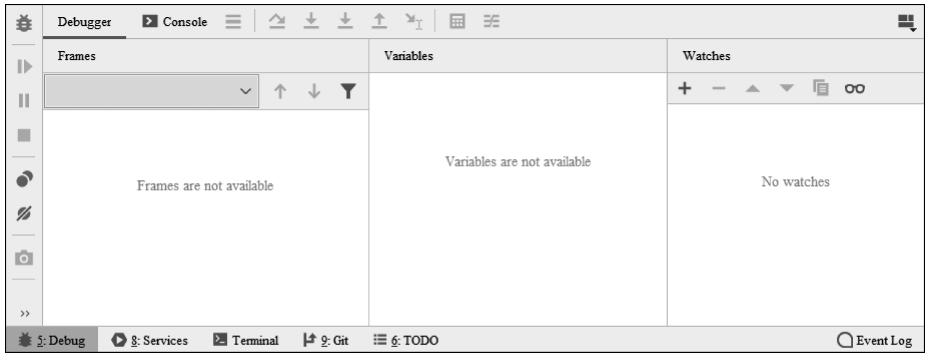

图 5.15 Debug 调试窗口

断点是进行 Debug 调试的前提，它标示了调试的具体位置。当程序运行到断点之后会自动停止并交由用户来完成后续的执行步骤，通过在调试过程中随时观察与修改具体的参数以发现异常位置及产生原因。

所以在使用 Debug 方式进行调试的时候至少需要一个断点。如果以 Debug 方式运行的项目没有设置任何断点，则它与运行模式的效果是一致的，只有添加了断点才可以进行具体的调试工作。

5.3.2　按钮与快捷键

接下来对 Debug 窗口进行说明。

1. 服务按钮

在 Debug 调试窗口中最左侧是服务按钮组，它们主要负责关闭/启动服务，以及设置断点等，各按钮功能如下：

(1) 按钮是 Debug 模式的启动按钮，在程序停止后可再次单击此按钮启动 Debug 模式运行。在启动之后它会变为按钮，单击此按钮后程序会自动跳过所有的断点重新运行并且在第一个遇到的断点处停止。

(2) 按钮是 Debug 模式的恢复按钮，单击此按钮后程序将从中断状态恢复并继续向下执行直至遇到下一个断点或运行结束，它可以跳过当前的断点。

(3) 按钮是 Debug 模式的暂停按钮，单击此按钮后可以在程序当前运行位置发起暂停，当暂停发起时如果当前执行位置没有设置断点，则依然可以启动断点的调试功能或使用恢复按钮继续向下执行。由于行级程序单步执行速度比较快，因此很难捕捉到一个准确的执行时间点，但是对于线程等运行时间较长的任务比较有效。

(4) 按钮用于终止当前程序的运行，终止运行的程序可以通过按钮再次启动调试。

(5) 按钮用于查看程序中所有的断点，如图 5.16 所示。

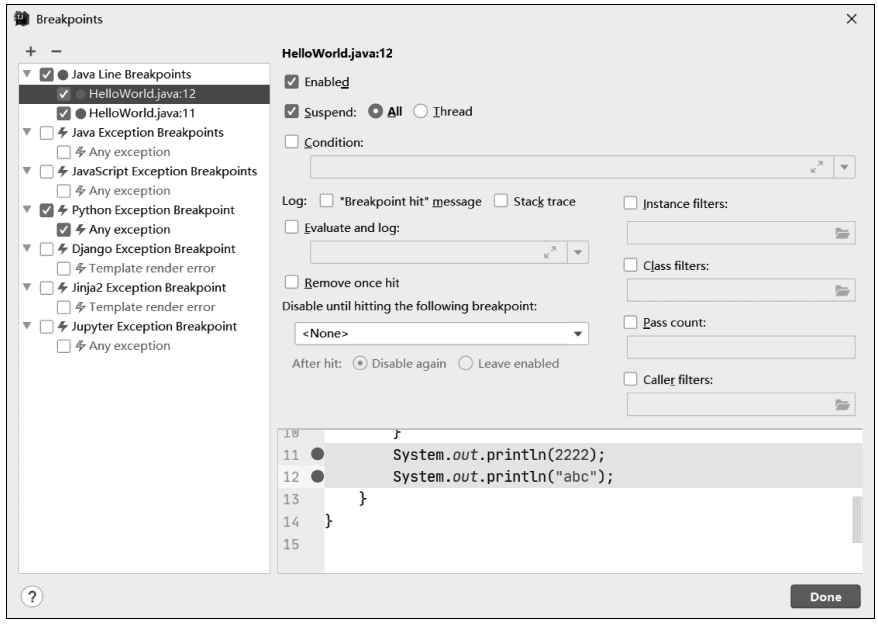

图 5.16　查看断点

(6) 按钮用于对断点执行静音操作,单击后会使所有未执行的断点在当前运行环境中失效从而让程序快速执行到最后,当再次运行调试时所有断点将会再次生效。

(7) 按钮可以获取当前系统运行的快照。如图 5.17 所示,当前程序只有主程序处于运行状态。

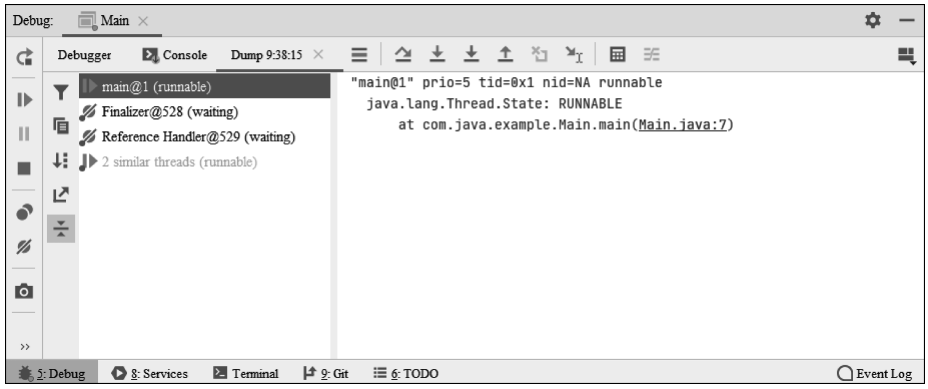

图 5.17 查看快照

(8) 按钮用于进行额外的操作,如显示行内值、恢复被静音的断点等,如图 5.18 所示。

(9) 按钮用于固定当前标签页。

2. 调试按钮

调试按钮组包含 8 个主要的操作按钮,主要用于断点调试过程中的追踪、跳跃等操作,如图 5.19 所示。

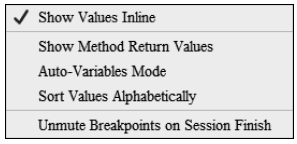

图 5.18 辅助操作

图 5.19 调试按钮

1) 跳转执行点(Show Execution Point)

当调试窗口处于调试状态且运行未完成时,跳转执行点可以将光标快速定位到当前调试位置对应的代码行。如果编辑区中打开多个文件使得光标在其他行或其他文件中,则单击此按钮可快速找到正在调试的位置。

当然,用户还可以使用快捷键 Alt+F10 来快速跳转到调试位置。

2) 步过(Step Over)

单击此按钮可以将调试过程逐行向下执行,如果某行上有方法调用,则不会进入方法内部,其快捷键为 F8。

3）步入(Step Into)

当调试步骤执行到某一行时，如果当前行有方法调用，则会进入方法内部，这些方法通常为用户的自定义方法，不会进入官方类库的方法，其快捷键为 F7。

4）强制步入(Force Step Into)

强制步入与步入一致且其图标为红色，当调试步骤执行到某一行时可以强制进入任何方法，此操作对于调试查看框架底层源码或官方类库十分有用，也是对步入的功能补充，其快捷键为 Alt＋Shift＋F7。

5）步出(Step Out)

当执行步出操作时会将步入的方法执行完毕，然后退出到方法调用处，只是还没有完成赋值，其快捷键为 Shift＋F8。

6）丢帧(Drop Frame)

在进行调试操作时，如果不小心跳过了某一步骤，则可以采用丢帧操作来“回退”到之前的堆栈帧并再次对该步骤进行调试，这相当于对时间进行追溯。

丢帧操作并不是一种真实的回退，它不会撤销对全局状态(如静态变量)进行的更改，只会重置局部变量，这可能会带来对应步骤二次调试的差异化。

7）运行至光标(Run to Cursor)

用户可以将光标定位到指定行，使用这个功能时代码会运行至光标行处且不需要打断点。

8）表达式计算(Evaluate Expression)

此操作可以在调试过程中计算某个表达式的值，而不用再去打印信息。当使用此功能时，单击“计算”按钮或使用快捷键 Alt＋F8 打开表达式计算窗口，如图 5.20 所示。

图 5.20　表达式计算窗口

在弹出的表达式计算窗口中输入选中某个表达式或变量，单击 Evaluate 执行计算，如图 5.21 所示。

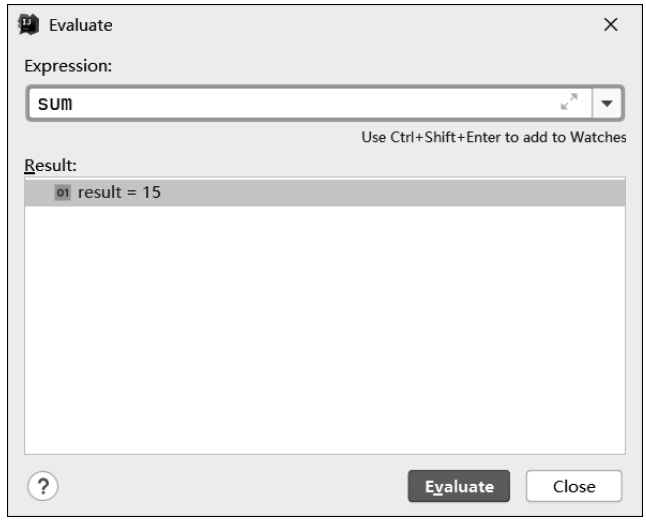

图 5.21　计算变量表达式

当然，也可以先选中某个表达式再使用快捷键 Alt＋F8，此时表达式或变量被自动填充到表达式区域。

值得说明的是，表达式不仅可以是一般变量或参数，还可以是方法。当代码中调用了方法时，可以通过这种方式查看某种方法的返回值，如图 5.22 所示。

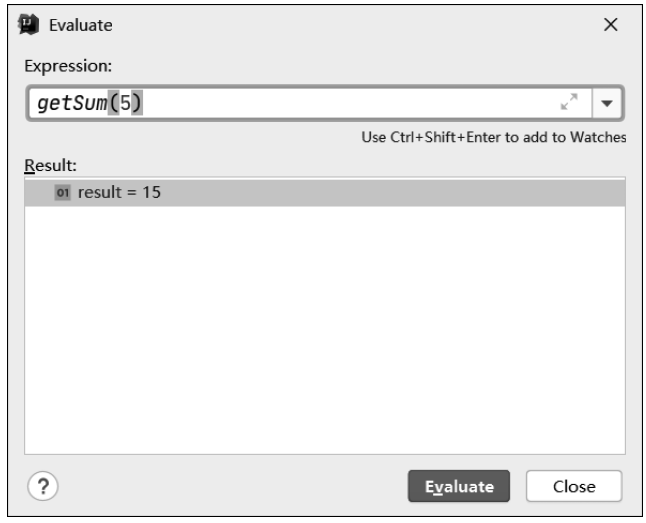

图 5.22　计算方法表达式

表达式计算窗口还可以动态地设置某些变量或结果的值，这对于依赖性计算十分有帮助，在对某些值进行动态调整后就可以按照不同的预期进行各种不同的调试。如将变量sum的值动态更改为20，就可以按图5.23所示的方式设置。

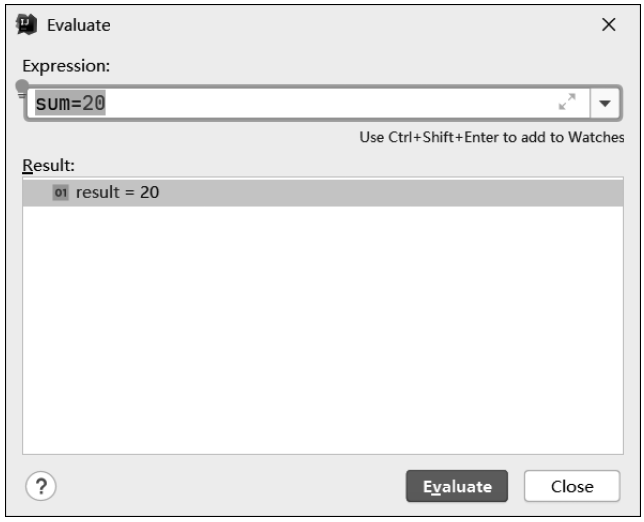

图5.23　修改表达式的值

在通过赋值运算符调整表达式的值后按Enter键，新的值就被注入相应的变量或方法中。

3. 查看变量

在调试过程中开发者需要对变量进行跟踪以观察程序运行是否达到了预期，IntelliJ IDEA中提供了多种查看变量的方式。

在进行Debug调试时，IntelliJ IDEA会在代码的行末显示出已经计算出来的表达式的值。这种方式对于值的观察比较直观，如图5.24所示。

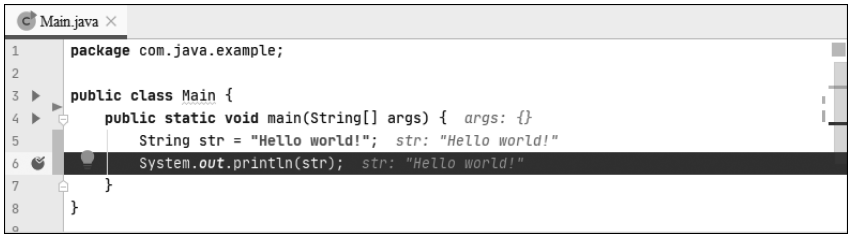

图5.24　值的直观显示

上述显示方式对于赋值类计算比较直接，但由于某些操作并不是赋值操作，如深度调用，因此这些值无法直接显示。

如果需要观察某些参数的变化，则可以将光标悬停到参数上，此时IntelliJ IDEA也会显示当前变量(或对象)的信息，如图5.25所示。

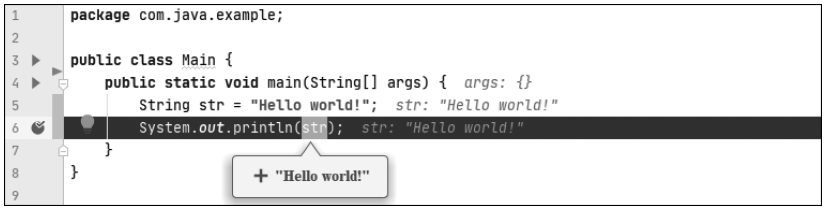

图 5.25　值的悬停显示

IntelliJ IDEA 提供了 Variables 变量窗口来查看所有变量的信息，如图 5.26 所示。

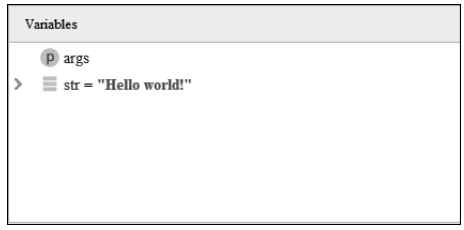

图 5.26　变量窗口

如果 Variables 窗口中显示变量信息过多，则对其查看也极为不便。IntelliJ IDEA 提供了 Watches 窗口来辅助开发者具体查看某个感兴趣的变量信息。

如果无法找到 Watches 窗口，则读者可以单击 Debugger 标签窗口右上方的视图按钮选择需要的视图窗口进行展示，如图 5.27 所示。

在低版本的 IntelliJ IDEA 上可能并没有图 5.27 所示的窗口查看按钮，但是在 Variables 窗口左侧有一系列图标按钮。当单击 ∞ 按钮时会单独打开 Watches 窗口，同时 Variables 窗口左侧的所有图标迁移到 Watches 窗口中。

用户可以从 Variables 窗口里拖曳选择的变量到 Watches 窗口里进行查看，也可以单击 New Watch 按钮在 Watches 窗口下方添加一个文本输入区域，用户在其中输入需要查看的变量后按 Enter 键即可，如图 5.28 所示。

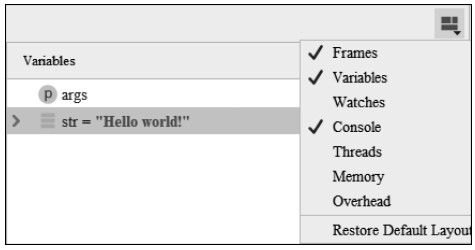

图 5.27　显示视图窗口

图 5.28　查看变量

右击待查看的变量，单击弹出菜单 Customize Data Views 打开定制数据视图窗口，如图 5.29 所示。

找到 Enable alternative view for Collections classes 交互视图选项，建议取消勾选此选项，因为它会引发很多对集合操作时的异常问题。为了观察方便先保持其勾选状态，然后通过实例演示存在哪些异常问题。

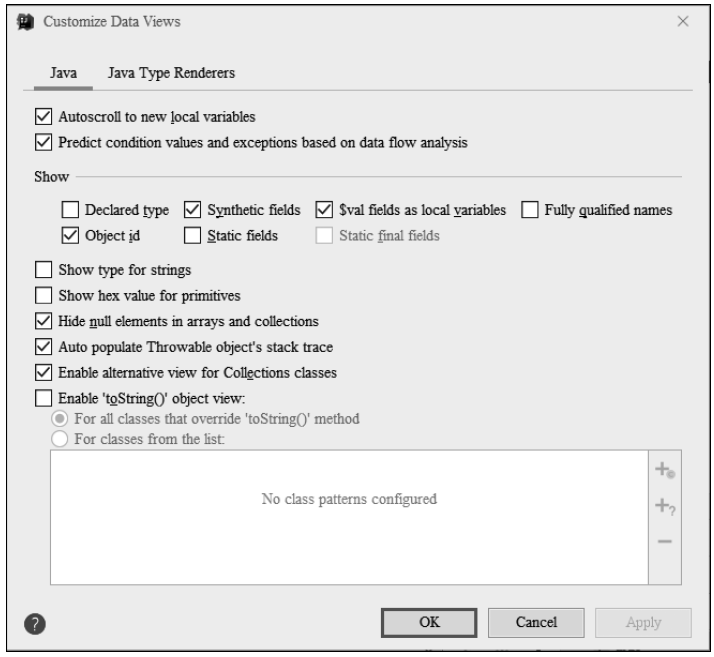

图 5.29　定制数据视图

首先以 Debug 模式运行示例程序,代码如下:

```
//第 5 章/ConcurrentLinkedQueueTest.java
import java.lang.reflect.Field;
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentLinkedQueueTest {
    public static void main(String[] args) {
        ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<>();
        print(queue);
        queue.offer("item1");
        print(queue);
        queue.offer("item2");
        print(queue);
        queue.offer("item3");
        print(queue);
    }

    private static void print(ConcurrentLinkedQueue queue) {
        Field field = null;
        boolean isAccessible = false;
        try {
            field = ConcurrentLinkedQueue.class.getDeclaredField("head");
```

```
                isAccessible = field.isAccessible();
                if (!isAccessible) {
                    field.setAccessible(true);
                }
                System.out.println("head: " + System.identityHashCode(field.get(queue)));
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                field.setAccessible(isAccessible);
            }
        }
    }
```

正常情况下,无论是在 Run 模式运行还是在 Debug 无断点模式运行,输出的每个 head 值都是相同的,Run 模式运行输出结果如下:

```
head: 356573597
head: 356573597
head: 356573597
head: 356573597
```

Debug 模式运行输出结果如下:

```
head: 766572210
head: 766572210
head: 766572210
head: 766572210
```

当在每个输出位置添加断点后,Debug 模式运行输出结果如下:

```
head: 766572210
head: 1020391880
head: 1020391880
head: 1020391880
```

为什么会这样? 还记得在使用 Debug 模式运行时 IntelliJ IDEA 界面上显示的变量值吗? 没错,为了显示这些变量的信息,IntelliJ IDEA 会调用对象的 toString()方法。虽然在正常情况下不会有任何影响,但是在 Debug 模式下它可能会带来意想不到的结果。

当调用 ConcurrentLinkedQueue 类的 toString()方法时会获取队列的迭代器,而创建迭代器时会调用队列的 first 方法,在 first 方法里会修改 head 的属性,从而导致输出的结果不一致。

为了不影响调试结果,需要关闭 IntelliJ IDEA 在 Debug 模式下的 toString()特性预览。执行菜单 File→Settings→Build,Execution,Deployment→Debugger→Data Views→Java 命令打开数据预览窗口,如图 5.30 所示,它与图 5.29 所示的配置其实是相同的。

取消勾选 Enable alternative view for Collections classes 和 Enable 'toString()' object

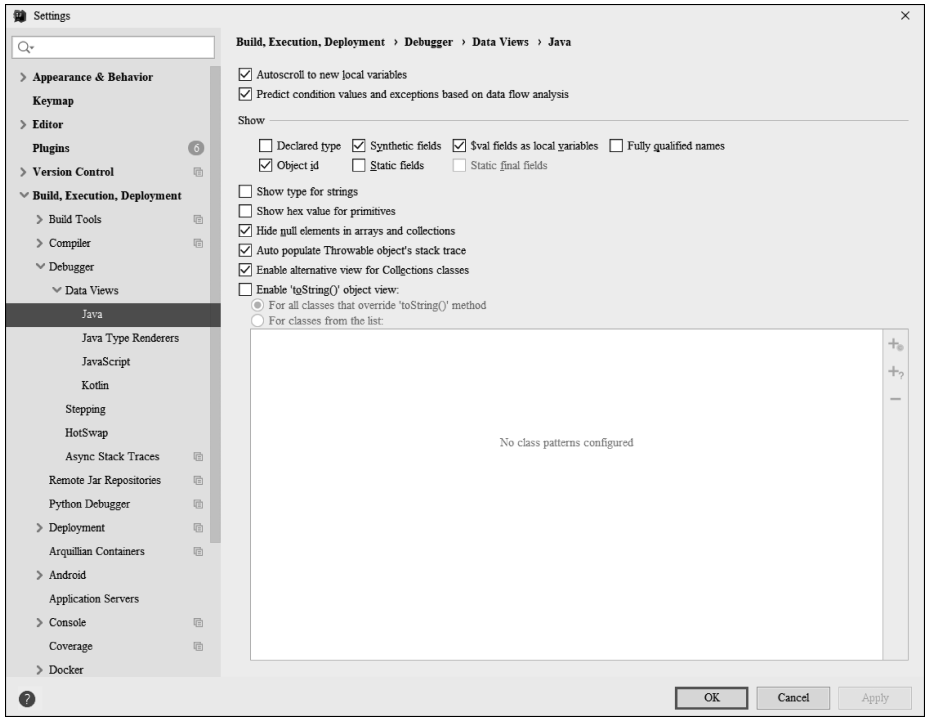

图 5.30　数据预览窗口

view 选项，保存后再次执行 Debug 调试并观察输出结果，此时可以看到输出结果正常了。

因为 ConcurrentLinkedQueue 是一个集合，选项 Enable alternative view for Collections classes 会在 Debug 调试时造成队列迭代器的遍历，只有把这两个特性一同关掉才会真正解决上面的问题。

5.3.3　设置断点条件

在进行调试时某些操作可能是在递归或遍历中进行条件判断的，在这种情况下很难控制进入指定的层次。通过设置断点条件以在满足条件时停在断点处，可以更有效地调试。

设置断点条件十分简单，右击需要设置条件的断点便可弹出条件设置窗口，如图 5.31 所示。

例如，在循环变量 i 为 3 时停在断点处，此时就可以按图 5.32 所示设置。

带有条件的断点图标会变为 ，当程序运行到断点处时图标会变为 。还可以在断点管理窗口设置断点执行条件，使用快捷键 Ctrl＋Shift＋F8 打开断点管理窗口，如图 5.33 所示。

在图 5.33 中勾选 Log 选项会将当前执行的断点行信息输出到控制台，这对于定位程序运行位置十分有帮助。勾选 Evaluate and log 选项可以在执行代码时计算表达式的值并将结果输出到控制台。

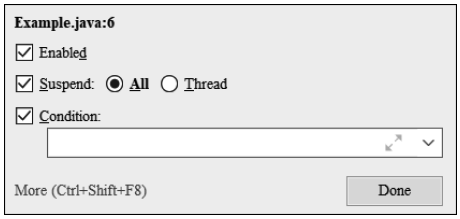

图 5.31　设置断点条件(一)

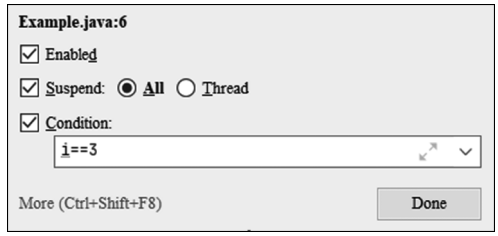

图 5.32　设置断点条件(二)

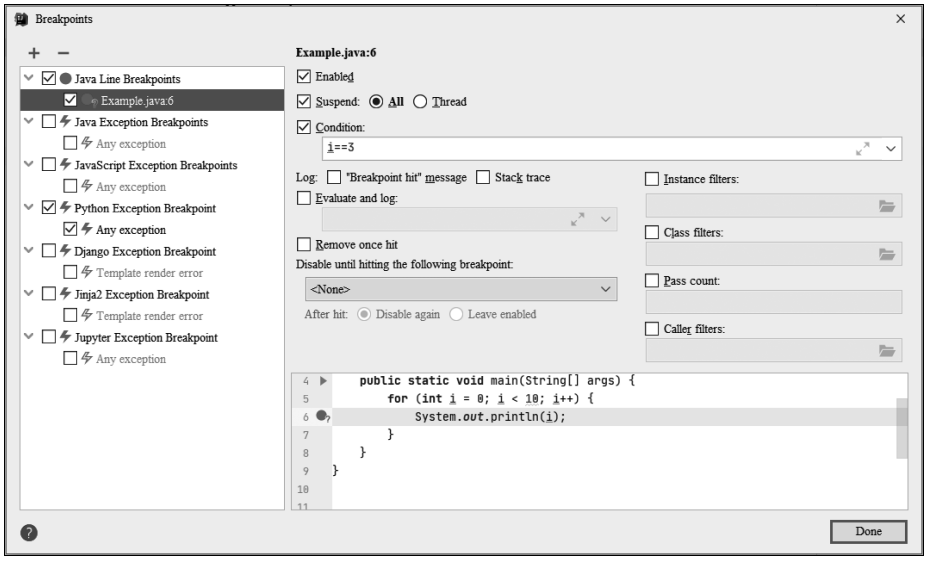

图 5.33　设置断点条件(三)

5.3.4　多线程调试

当进行某个 Debug 调试运行时可能需要发起另一个 Debug 调试,因为当前的 Debug 调试还没有停止,因此 IntelliJ IDEA 会弹出提示对话框并由用户来选择是否终止当前正在运行的调试,如图 5.34 所示。

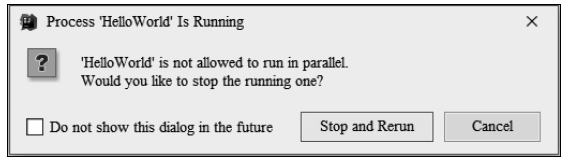

图 5.34　Debug 阻塞

IntelliJ IDEA 在进行 Debug 调试时默认阻塞级别是 ALL，因此会阻塞其他线程而导致无法再次发起调试，只有在当前调试线程运行完或终止时才可以发起其他调试线程。

在图 5.33 中可以观察当前调试的阻塞级别，为了能够以多线程的方式运行调试，需要将 Suspend 的值更改为 Thread。勾选后会显示 Make Default 按钮，直接确认后将多线程方式设置为默认方式，如图 5.35 所示。

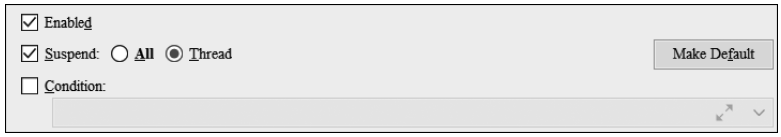

图 5.35　修改 Debug 阻塞级别

5.4　远程调试

IntelliJ IDEA 提供了对远程调试的支持，通过远程调试可以直接跟踪远程服务器上的程序运行状态并快速定位故障原因。虽然使用远程调试是直接对服务器上的应用进行定位，但是远程调试可能造成服务器上的请求阻塞，因为请求都被切换到了本地调试。

在进行远程调试时，远程服务器需要支持调试请求的连接。以 Linux 系统上的 Tomcat 服务器为例，其需要在 catalina.bat 文件中设置 JVM 相关运行的参数变量，参数设置如下：

```
export JAVA_OPTS = '-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000'
```

在远程服务启动后访问浏览器可以看到响应页面可正常显示，如图 5.36 所示。

图 5.36　远程服务已运行

为了与远程服务器进行连接，需要在 IntelliJ IDEA 的运行/调试配置窗口添加远程连接配置。单击 + 按钮新建 Remote 远程配置，如图 5.37 所示。

接下来打开远程配置详情，填写远程连接的名称、Host 地址及调试端口，此端口不同于服务器端口，如图 5.38 所示。

Debugger mode 用于指定调试的模式，其默认值为 Attach to remote JVM。Debugger mode 各配置含义如下。

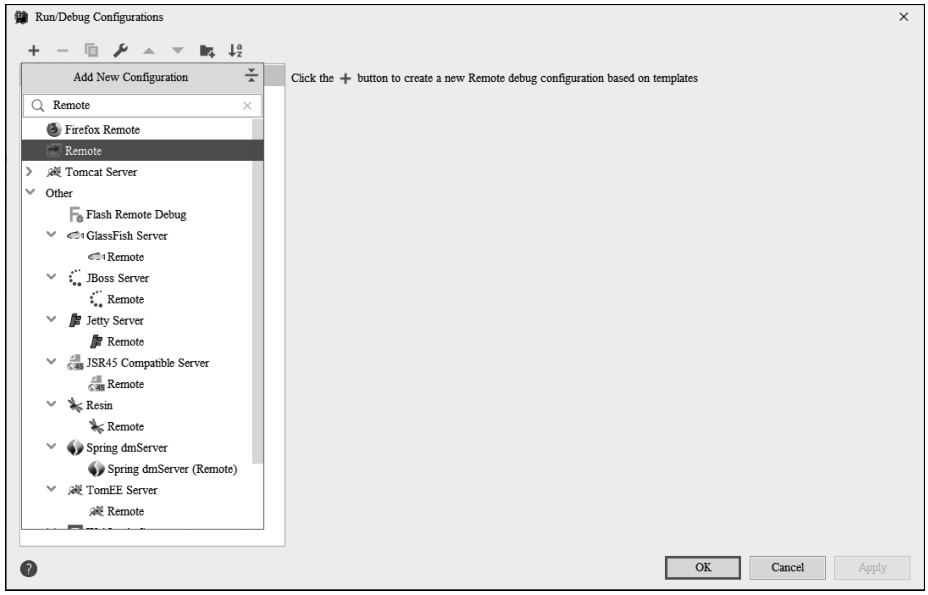

图 5.37　新建远程连接配置

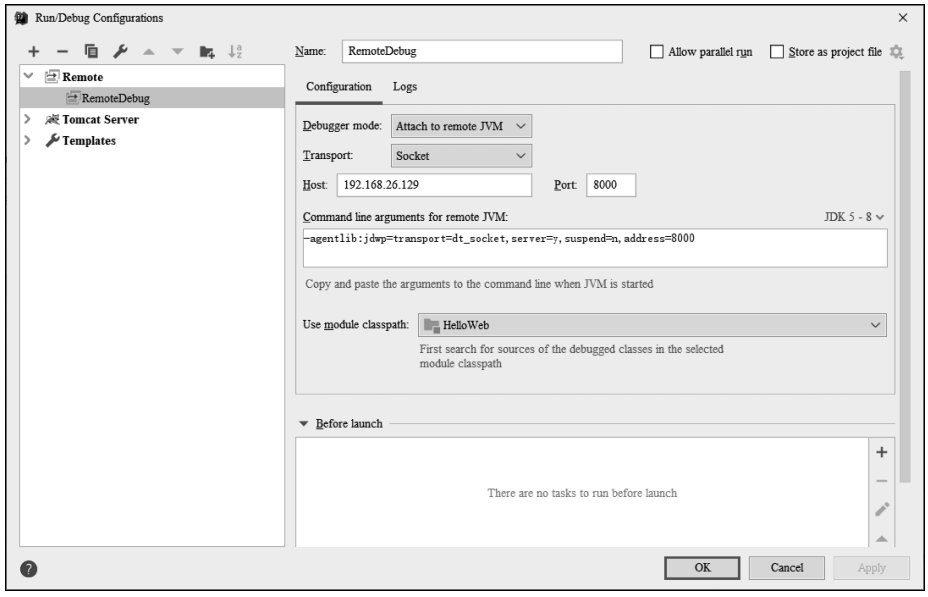

图 5.38　远程调试配置

(1) Attach to remote JVM：此模式下调试服务器端(被调试远程运行的服务)启动一个端口等待调试客户端的连接。

(2) Listen to remote JVM：此模式下调试客户端负责监听端口，当调试服务器端准备完成后会进行连接。

Transport 用于设置传输方式，其默认值为 Socket。Transport 下的配置含义如下。

(1) Socket：macOS 及 Linux 系统使用此种传输方式。

(2) Shared memory：Windows 系统使用此种传输方式。

配置完成后工具栏会添加对应的调试配置，如图 5.39 所示。

图 5.39　远程调试配置

单击调试按钮启动远程服务器连接，连接成功后会显示如图 5.40 所示的连接信息。

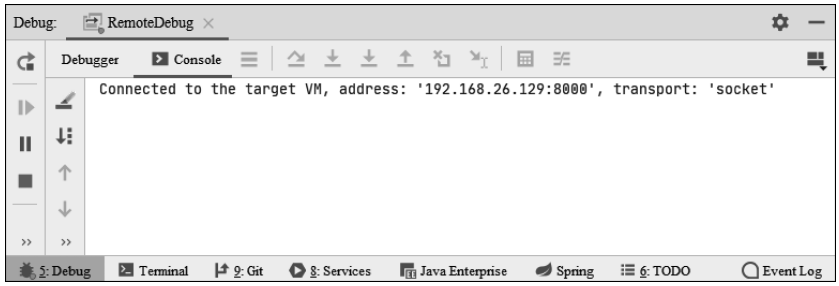

图 5.40　远程调试连接成功

在本地程序中添加断点，断点在 Debug 调试模式运行前后添加均可生效，如图 5.41 所示。

图 5.41　添加断点

在浏览器中访问请求对应的网址，发现浏览器响应处于响应等待状态，如图 5.42 所示。

图 5.42　响应等待

本地 IntelliJ IDEA 调试窗口成功接收请求并执行到断点位置，如图 5.43 所示。

此时可以进行 Debug 模式下的远程调试工作了。

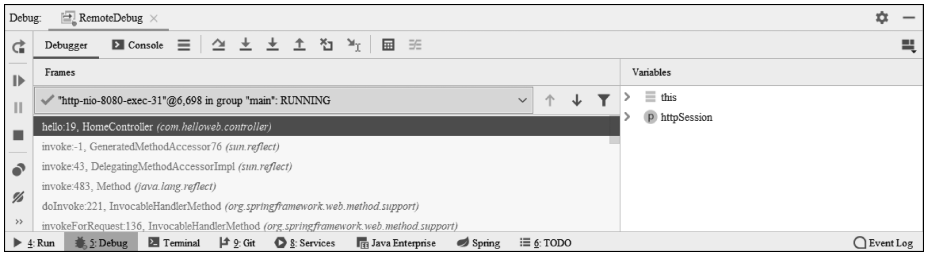

图 5.43　开始远程调试

5.5　本章小结

项目中一定会有某些错误或缺陷，这些错误和缺陷会在不同的时机与条件下出现。掌握项目调试与运行的技巧，可以快速准确地分析出问题出现的原因并及时排除影响项目稳定运行的潜在因素。