https://blog.csdn.net/swebin/article/details/70598978

**Eclipse的Debug调试技巧大全(精华版)**

2017年04月24日 16:13:17 [webin](https://me.csdn.net/swebin" \t "_blank) 阅读数 6628

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/swebin/article/details/70598978

**一、Debug视图**

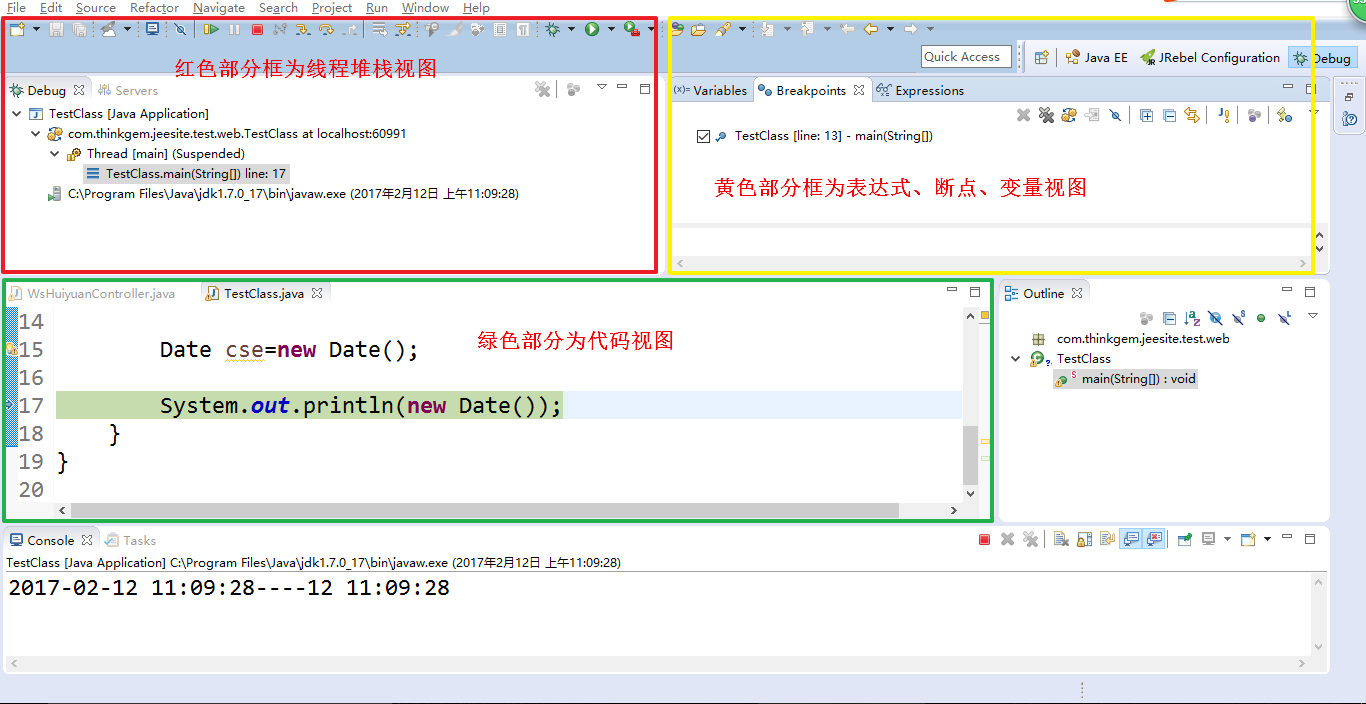
调试中最常用的窗口是:

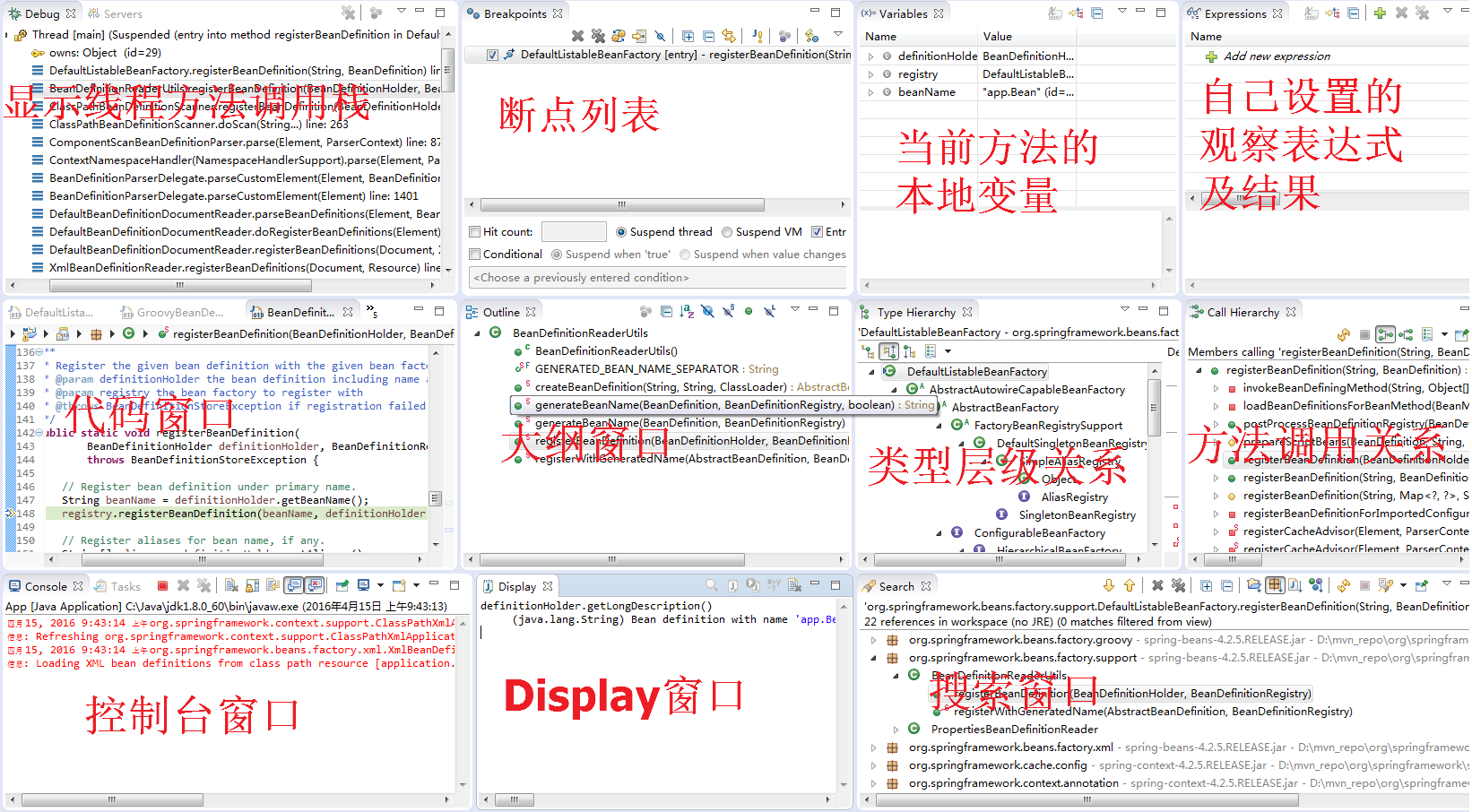
| **窗口** | **说明** |
| --- | --- |
| Debug窗口 | 主要显示当前线程方法调用栈， 以及代码行数(有调试信息的代码) |
| 断点Breakpoints窗口 | => 断点列表窗口，可以方便增加断点，设置断点条件，删除断点等 |
| 变量Variables窗口 | => 显示当前方法的本地变量，非static方法，包含this应用，可以修改变量值 |
| 代码编辑窗口 | => 这个不用多说了 |
| 输出Console窗口 | => 日志等输出内容，调试时，可以将关注的组件级别设置低一点，以便获得跟多输出信息 |

另外辅助的窗口有:

| **窗口** | **说明** |
| --- | --- |
| 表达式expression窗口 | => 写上自己需要观察的数据的表达式，或者修改变量值 |
| Display窗口 | => 可以在display中执行代码块，输出内容等 |
| 大纲Outline窗口 | => 查看当前类的方法，变量等 |
| 类型层级Type hierarchy窗口 | => 查看当前所在类的继承层次，包括实现接口，类继承层次 |
| 方法调用关系Call hierarchy窗口 | => 查看当前方法被哪些方法调用，调用方法在哪些类中、第几行，可以直接打开对应的方法 |
| 搜索结果Search窗口 | => 结合快捷键可以查看变量、方法等在工作空间、项目、工作集中被引用或定义的代码位置 |

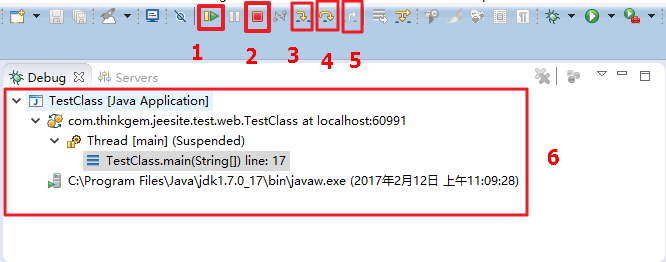
**1）窗口全览：**





**2）Debug View(线程堆栈视图):**

debug视图允许您在工作台上管理正在调试和运行的程序，他显示了你正在调试的程序中挂起的线程的堆栈帧，程序中的每个线程作为树的节点出现。他展示了正在运行的每个目标的进程。如果线程被挂起，它的堆栈帧以子元素的形式展示。以下是一些常用的debug按钮：



1.表示当前实现继续运行直到下一个断点，快捷键为F8。  
  
2.表示打断整个进程  
  
3.表示进入当前方法，快捷键为F5。  
  
4.表示运行下一行代码，快捷键为F6。  
  
5.表示退出当前方法，返回到调用层，快捷键为F7。  
  
6.表示当前线程的堆栈，从中可以看出在运行哪些代码，并且整个调用过程，以及代码行号

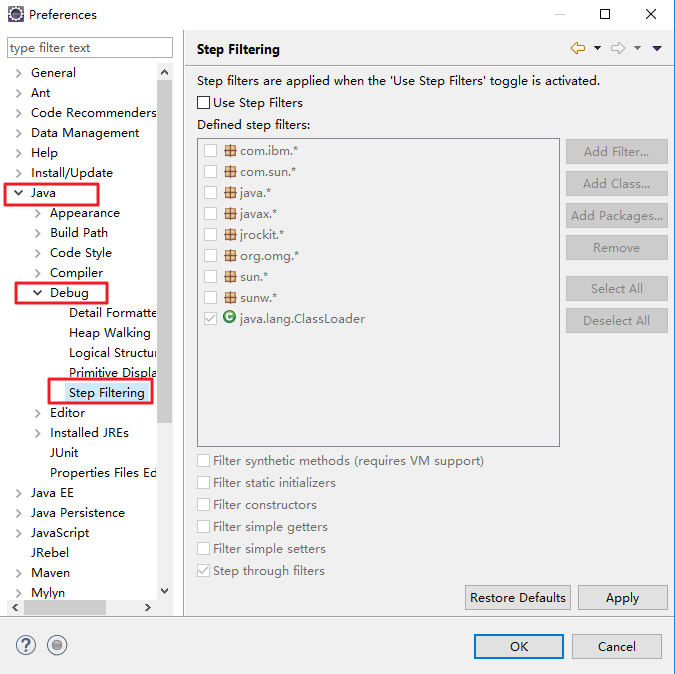
**详细介绍：**

https://img-blog.csdn.net/20170212113123026 Skip All Breakpoints : 将所有断点设置为被跳过的，设置了Skip All Breakpoints之后，所有断点上都会有一个斜线，表示断点将被跳过，线程不会在该断点处被挂起。

https://img-blog.csdn.net/20170212113228793 Drop to Frame : 这个命令可以让程序回到当前方法的开头第一行重新开始执行，可以重新执行这个**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "_blank)**堆栈帧，可以选择一个指定的堆栈帧，然后点击 Drop to Frame，这样就可以重新进入指定的堆栈帧。使用Drop to Frame时候需要注意：

1.不能drop到已经执行过的方法栈中的方法中。  
  
2.drop到stack frame中时，不会改变全局数据原有的值，比如，一个包含元素的vertor并不会被清空。

https://img-blog.csdn.net/20170212113338857 Step Filters : 这个功能比较简单，就是当我们在debug的时候想要忽略一些我们不关注的类时，可以开启Step Filters进行过滤，程序会一直执行直到遇到未经过滤的位置或断点。Step Filters功能由Use Step Filters，Edit Step Filters，Filter Type，Filter Package四项组成。具体操作如下：

步骤 1： Windows -> Preferences -> Java -> Debug -> Step Filtering.  
  
  
  
步骤 2：选择‘Use Step Filters’  
  
步骤 3：在屏幕上选中所需的选项。你可以添加你自己代码库中的部分代码。  
  
步骤 4：点击‘Apply’

原理上，Edit Step Filter命令用于配置Step Filter规则，而Filter Type与Filter Package分别指的是过滤的Java类型与**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "_blank)**Package。

https://img-blog.csdn.net/20170212114352676 Step Return : 跳出当前方法，在被调用方法的执行过程中，使用Step Return会在执行完当前方法的全部代码后跳出该方法返回到调用该方法的方法中。

https://img-blog.csdn.net/20170212114510264 Step Over : 在单步执行时，在函数内遇到子函数时不会进入子函数内单步执行，而是将子函数整个执行完在停止，也就是把子函数整个作为一步。

https://img-blog.csdn.net/20170212114551812 Step Into：单步执行，遇到子函数就进入并且继续单步执行

https://img-blog.csdn.net/20170212114627234 Resume：恢复暂停的线程，直接从当前位置跳到下一个断点位置。

https://img-blog.csdn.net/20170212114657032 Suspend：暂停选定的线程，这个时候可以进行浏览或者修改代码，检查数据等。

Eclipse通过Suspend与Resume来支持线程的暂挂与恢复。一般来讲，Suspend适用于多线程程序的调试，当需要查看某一个线程的堆栈帧及变量值时，我们可以通过Suspend命令将该线程暂挂。Resume用于恢复。

有两种Resume需要注意：

第一是当在调试过程中修改程序代码，然后保存，点击Resume，此时程序会暂挂于断点。

第二是当程序抛出异常时，运行Resume，程序也会暂挂于断点。

https://img-blog.csdn.net/20170212114815096 Terminate : Eclipse通过Terminate命令终止对本地程序的调试。

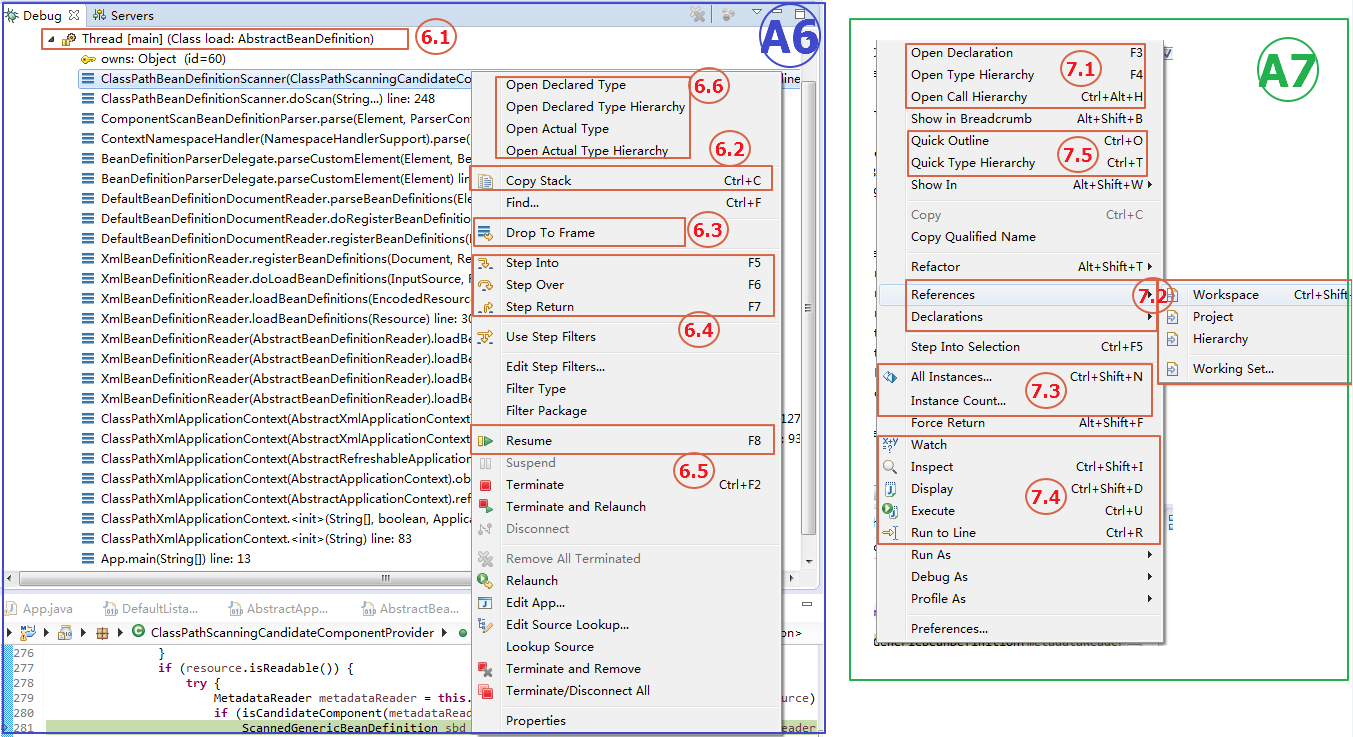
https://img-blog.csdn.net/20170212114854409 Disconnect:Eclipse使用Disconnect命令来终止与远程JVM的socket连接。

**1、调试执行**

| **标记** | **功能** | **快捷键** | **描述** |
| --- | --- | --- | --- |
| 6.4 | Step Info | F5 | -> 单步进入(如果有方法调用，将进入调用方法中进行调试); |
| 6.4 | Step Over | F6 | -> 单步跳过(不进入行的任何方法调用中，直接执行完当前代码行，并跳到下一行); |
| 6.4 | Step Return | F7 | -> 单步返回(执行完当前方法，并从调用栈中弹出当前方法，返回当前方法被调用处)； |
| 6.5 | Resume | F8 | -> 恢复正常执行(直到遇到下一个断点); |
| 7.4 | Run to Line | Ctrl+R | -> 执行到当前行(将忽略中间所有断点，执行到当前光标所在行); |
| 6.3 | Drop To Frame | 无 | -> 回退到指定方法开始处执行，这个功能相当赞。 在方法调用栈上的某个方法右键，选择Drop To Frame就可以从该方法的开始处执行 比如重新执行本方法，可以在本方法上用Drop To Frame，将从本方法的第一行重新执行。 当然对于有副作用的方法，比如数据库操作，更改传入参数的对象内容等操作可能重新执行就再是你想要的内容了。 |
| 6.1+6.2 | Copy Stack | 无 | -> 拷贝当前线程栈信息 |

如果在调试时，需要排除一些类、包等不需要进入调试，可以使用Edit Step Filters设置。

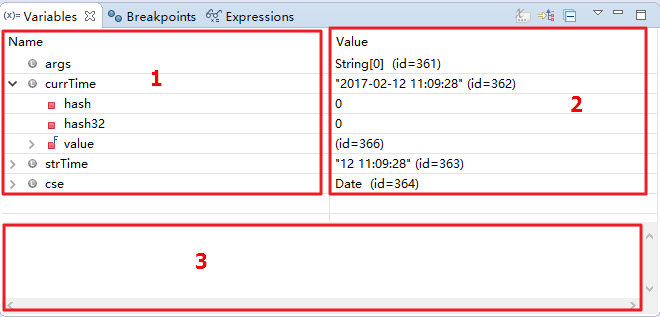
A6的properties： Java进程启动相关信息，包括控制台启动参数，环境参数等。如果参数启动参数有问题可以先看看这里的实际启动参数是不是有误。另外还可以查看虚拟机支持调试的相关选项。



**2、数据查看**

| **标记** | **功能** | **快捷键** | **描述** |
| --- | --- | --- | --- |
| 7.4 | Inspect | ctrl+shift+i | -> 察看选择的变量、表达式的值或执行结果, 再次按ctrl+shift+i可以将当前表达式或值添加到Expressions窗口中查看; |
| 7.4 | Display | ctrl+shift+d | -> 显示选择的变量、表达式的值或执行结果, 再次按ctrl+shift+d可以将当前表达式或值添加到Display窗口中显示; |
| 7.4 | Execute | ctrl+u | -> 执行选择表达式; |
| 7.4 | Run to Line | ctrl+r | -> 执行到当前行(将忽略中间所有断点，执行到当前光标所在行); |
| 7.3 | All Instances | ctrl+shift+n | -> 查看选择的类的所有对象，这个功能超赞； |
| 7.3 | Instance Count | 无 | -> 查看选择的类的所有对象个数； |
| 7.4 | Watch | 无 | -> 添加当前变量、表达式到Expressions窗口中； |

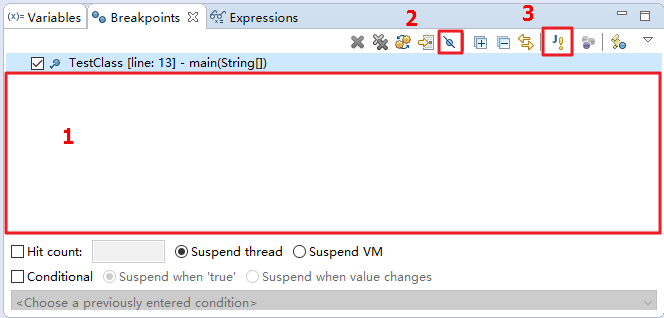
**3）Variables View (变量视图)**



1.为变量名视图，显示当前代码行中所有可以访问的实例变量和局部变量  
  
2.显示所有的变量值  
  
3.可以通过该窗口来改变变量值

Variables View显示与Debug View中选定的堆栈帧相关的变量信息，调试Java程序时，变量可以选择将更详细的信息显示在详细信息窗格中。此外，Java对象还可以显示出其包含的属性的值。在该窗口中选中变量鼠标右键点击可以进行许多操作，主要操作有以下这些：  
  
All Instances：打开一个对话框来显示该java类的所有实例，使用该功能需要java虚拟机支持实例的检索。  
  
All References::打开一个对话框来显示所有引用了该变量的java对象，  
  
Change Value：：更改变量的值，该功能可以和Drop to Frame联合使用进行程序的调试。使用这两个功能就可以代替重新debug  
  
Copy Variables：复制变量的值，尤其在变量值很长（比如json数据）的时候，这个功能就派上用场了。  
  
Find：有的时候一个类中变量特别多的时候，可以进行查找。

**4）Breakpoints View (断点视图)**

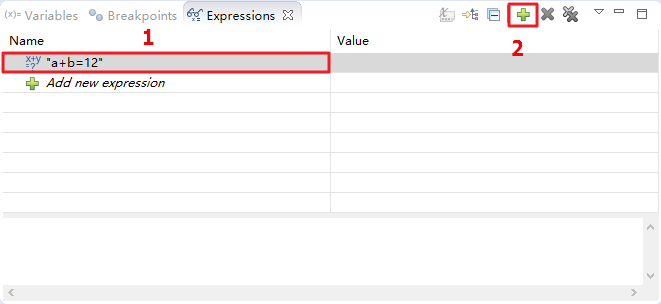


1.显示所有断点  
  
2. 将当前窗口1中选中的端口失效，再次点击启用。  
  
3.异常断点

Breakpoints View将列出你在当前工作区间里设置的所有断点，双击断点可以进入到程序中该断点的位置。还可以启用或禁用断点，删除，添加新的，根据工作组或点命中计数给他们分组。在使用断点是有以下两个技巧是十分有用的：  
  
Hit Count: 是指定断点处的代码段运行多少次，最典型的就是循环，如果要让一个循环执行10次就线程挂起，则指定Hit Count值为10，那么当前的循环执行到第九次的时候就会挂掉。  
  
Conditional:顾名思义，就是条件判断，例如我们需要循环变量i==10时，线程挂起，则条件设定为i==10,选择Suspend when “true”。

那如果上面的Hit Count和Conditional都选择的话，如果表达式和值设置不合理则会失效。如果选择Suspend when value changes，那么可能在Conditional在变量值发生改变的时候就挂起。

**5）Expressions View (表达式视图)**



1.表达式  
  
2. 点击此可以新增一个表达式

要在 Debug 透视图的编辑器中求表达式的值，选中设置有断点的一整行，并在上下文菜单中选择 Inspect 选项。表达式是在当前堆栈帧的上下文中求值的，其结果显示在 Display 窗口的 Expressions 视图中。 比如我想要计算变量a+b的值，那么就可以在表达式视图中加一个表达式：a+b

**6）Display View**

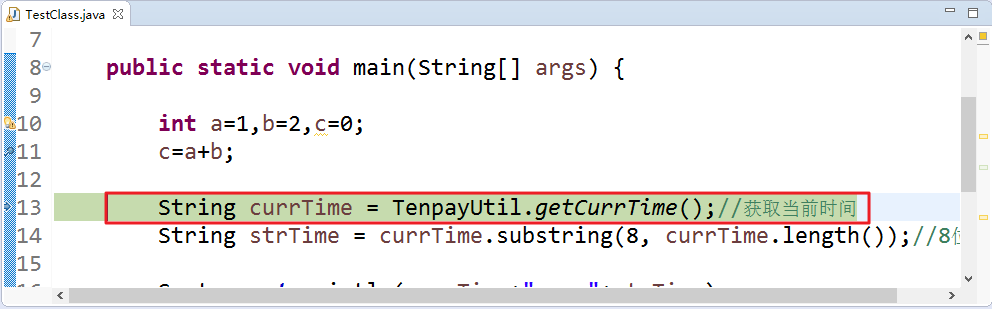


可以使用这个视图，输入或者演算一些新的代码。这些代码在当前的调试位置的上下文环境中被执行，这意味着，你可以使用所有变量甚至是内容助手。要执行你的代码的话，只需标记它，并使用右键菜单或者CTRL+U(执行)或者 CTRL+SHIFT+I (检查)

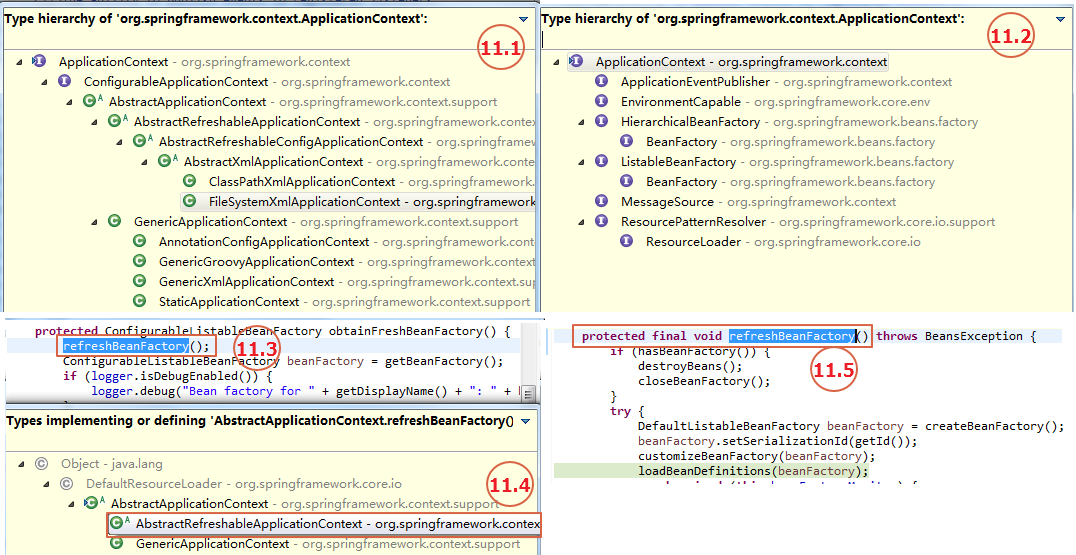
**7）代码查看辅助窗口**

**1、代码视图:**

代码视图,用来显示具体的代码。其中绿色部分是指当前将要执行的代码

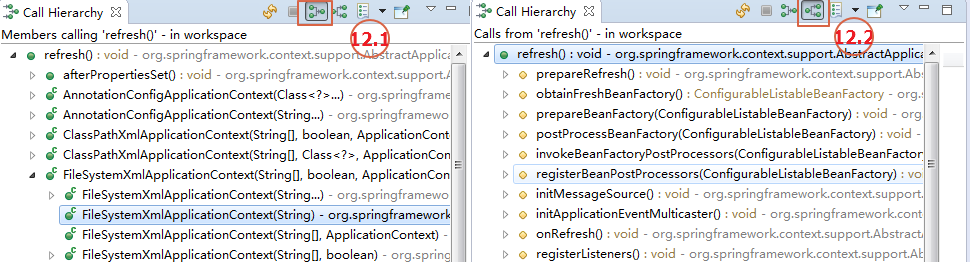


| **标记** | **功能** | **快捷键** | **描述** |
| --- | --- | --- | --- |
| 11.1~11.5 | quick type hierarchy | ctrl+t | 查看当前类、接口的继承层次， 默认进入时，显示继承/实现当前类/方法的子类，子接口11.1；再次ctrl+t，将显示当前类、接口继承/实现的超类/接口11.2；调试时，经常用该功能，在接口或抽象类的方法调用处11.3，ctrl+t察看实现类11.4，直接导航到对应的实现方法中11.5。 |
|  | quick outline | ctrl+o | 查看当前类的大纲，包括方法，属性等内容； 用处不大； |
|  | open declarations | F3 | 查看变量、属性、方法定义的地方 |



**2、Call Hierarchy窗口:**

| **标记** | **功能** | **快捷键** | **描述** |
| --- | --- | --- | --- |
| 12.1~12.2 | open call hierarchy | ctrl+alt+h | 查看方法被调用层次， 可以看当前方法被调用的地方12.1，或者当前方法调用了其他类的方法12.2 |

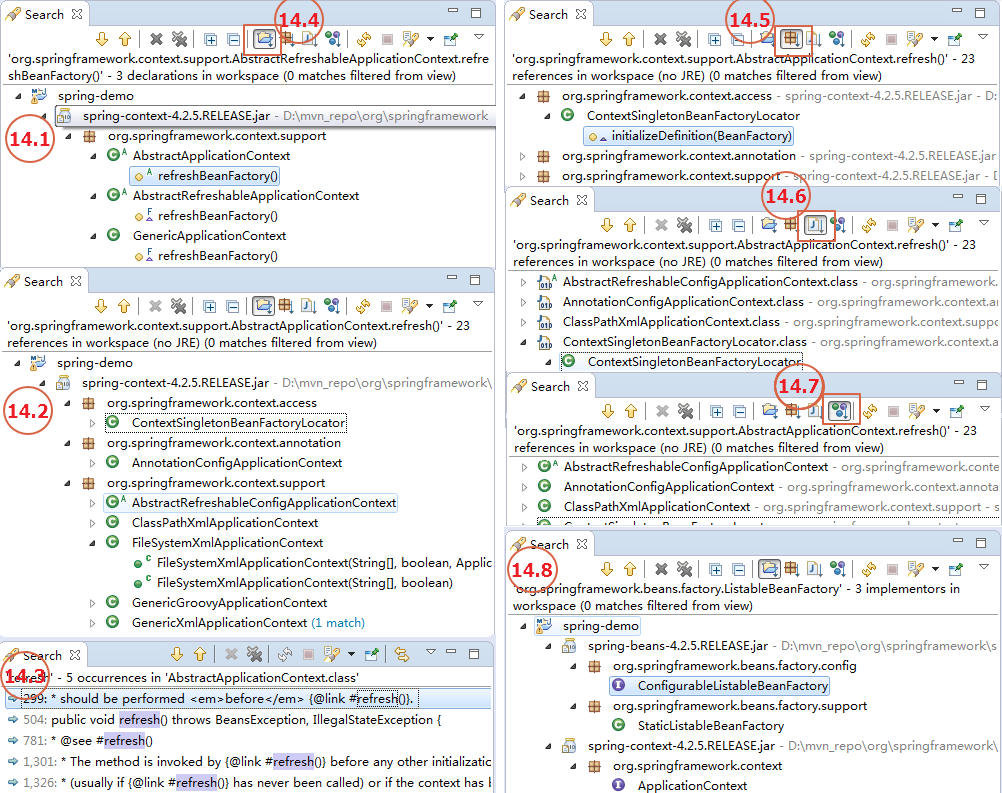


**3、Type Hierarchy窗口：**

| **标记** | **功能** | **快捷键** | **描述** |
| --- | --- | --- | --- |
| 13.1~13.4 | open type hierarchy | F4 | 查看继承层次， 可以查看类的继承层次，包括子类父类13.1， 或者类实现的接口继承层次13.2，还会根据选择的类/接口，在右边显示该类的大纲13.3、 13.4； 13.3可以选择是否显示父类/父接口的属性、方法等 |

**4、Search 窗口：**

| **标记** | **功能** | **快捷键** | **描述** |
| --- | --- | --- | --- |
| 14.1 | declarations | ctrl+g | 相同的方法签名在工作空间中及第三方jar包中被定义的位置14.1 |
| 14.2 | references | ctrl+shif+g | 当前选中的变量、属性、方法在工作空间中及第三方jar包中被引用的位置14.2 |
| 14.3 |  | ctrl+shift+u | 查看变量、属性 、方法在当前类中出现的地方14.3 |
| 14.4 | implements |  | 查看实现当前接口的类14.8 |
| 14.4~14.7 | 显示方式 |  | 可以选择不同的显示方式，具体看图 |



**二、Debug**

**1.设置断点**  
  
在源代码文件中，在想要设置断点的代码行的前面的标记行处，双击鼠标左键就可以设置断点，在相同位置再次双击即可取消断点。有的时候我们还有这样的需要，就是我并不想一行一行的执行代码，比如一个for循环会循环1000多遍，我只想在第500遍的时候让线程挂起进行调试，这个时候我们可以使用**条件断点。 设置条件断点：我们可以给该断点设置触发条件，一旦满足某条件是才开始调试，可以在断点处点击鼠标右键，选择Breakpoint Properties进入断点设置页面**，刚刚在讲断点视图的时候我们学到过Hit Count和Conditional的用法，这里可以设置条件和执行次数。

**1.1）断点类型及断点窗口**  
  
在调试中可以设置的断点类型有五种：

1.**行断点(line breakpoints)**: 条件断点，顾名思义就是一个有一定条件的断点，只有满足了用户设置的条件，代码才会在运行到断点处时停止。

2.**方法断点(method breakpoints )**: 方法断点的特别之处在于它可以打在 JDK的源码里，由于 JDK 在编译时去掉了调试信息，所以普通断点是不能打到里面的，但是方法断点却可以，可以通过这种方法查看方法的调用栈。

3.**观察断点(watch breakpoints-成员变量访问变更)**

4.**异常断点(exception breakpoints)**

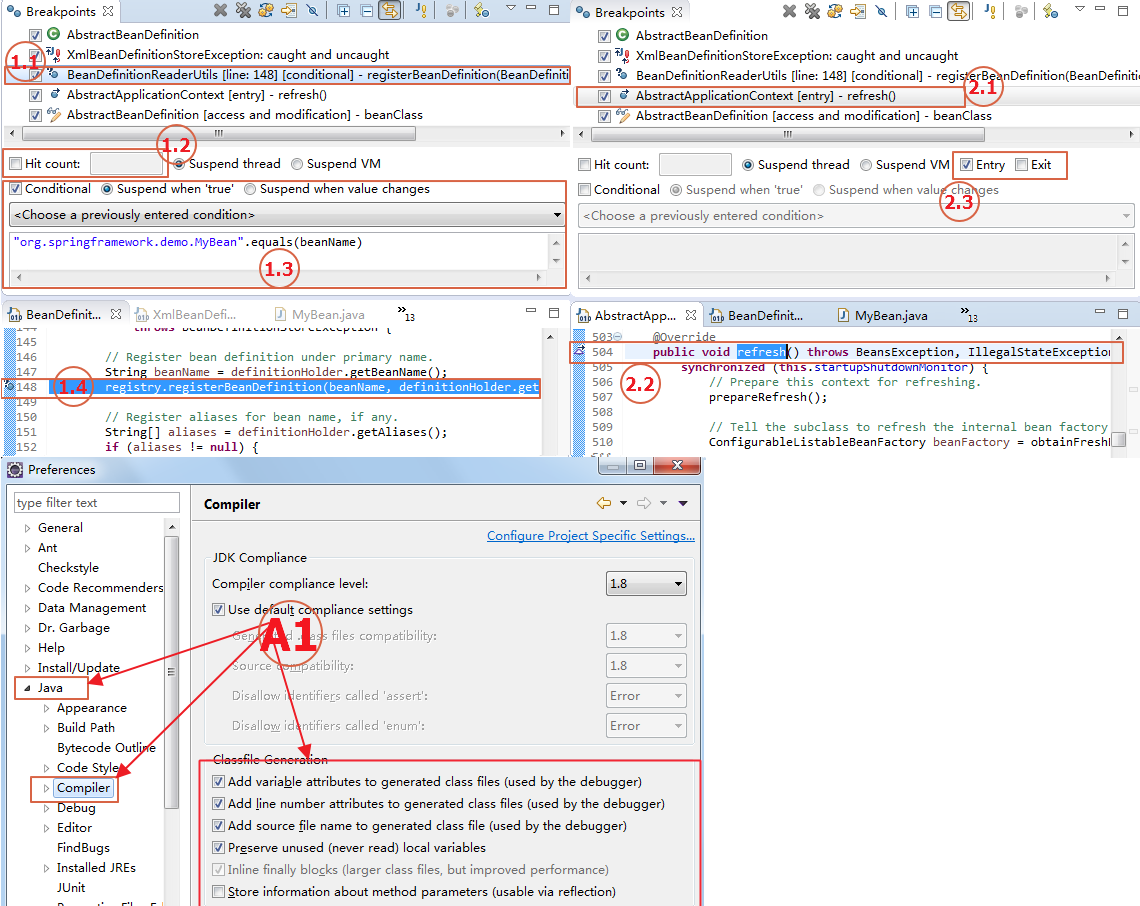
5.**类加载断点(class load breakpoints)**

每种断点的设置有些许不一样，可以在断点上右键->Breakpoint properties进行设置，但一般在断点窗口有快速设置的界面，Breakpoint properties中多了filter, 其实比较鸡肋，用处不大。

断点相关的快捷键：

| **快捷键** | **说明** |
| --- | --- |
| ctrl+shift+b | 在光标处大断点/取消断点 |
| ctrl+alt+b | 忽略所有断点 |
| Alt+shift+q, b | 激活断点窗口 |

1、行断点： 在方法中的某一行上打断点1.1、1.4。行断点可以设置挂起线程/VM的条件1.3，访问次数1.2。   
1.3中的条件是，**[spring](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "_blank)**在注册Bean定义(registerBeanDefinition)时，如果是org.springframework.demo.MyBean，就挂起线程，可以开始单步调试了。   
对于命中次数(hit count)1.2的使用，一般是在循环中，第N个对象的处理有问题，设置hit count = N， 重调试时，可以方便到达需要调试的循环次数时，停下来调试。  
  
  
2、方法断点：在方法上打断点2.1、2.2。方法断点的好处是可以从方法方法进入或者退出时2.3，停下来调试，类似行断点，而且只有行断点和方法断点有条件和访问次数的设置功能。   
但是方法断点还有另外一个好处，如果代码编译时，指定不携带调试信息，行断点是不起作用的，只能打方法断点。   
有兴趣的可以通过A1将Add line number… 前的勾去掉， 调试下看看。



3、观察断点： 在成员变量上打的断点3.1、3.3。只有对象成员变量有效果，静态成员变量不起作用。   
可以设置变量被访问或者设置的时候挂起线程/VM 3.2，也就是类似3.4的所有对成员变量的访问或者设置的方法都会被监控到

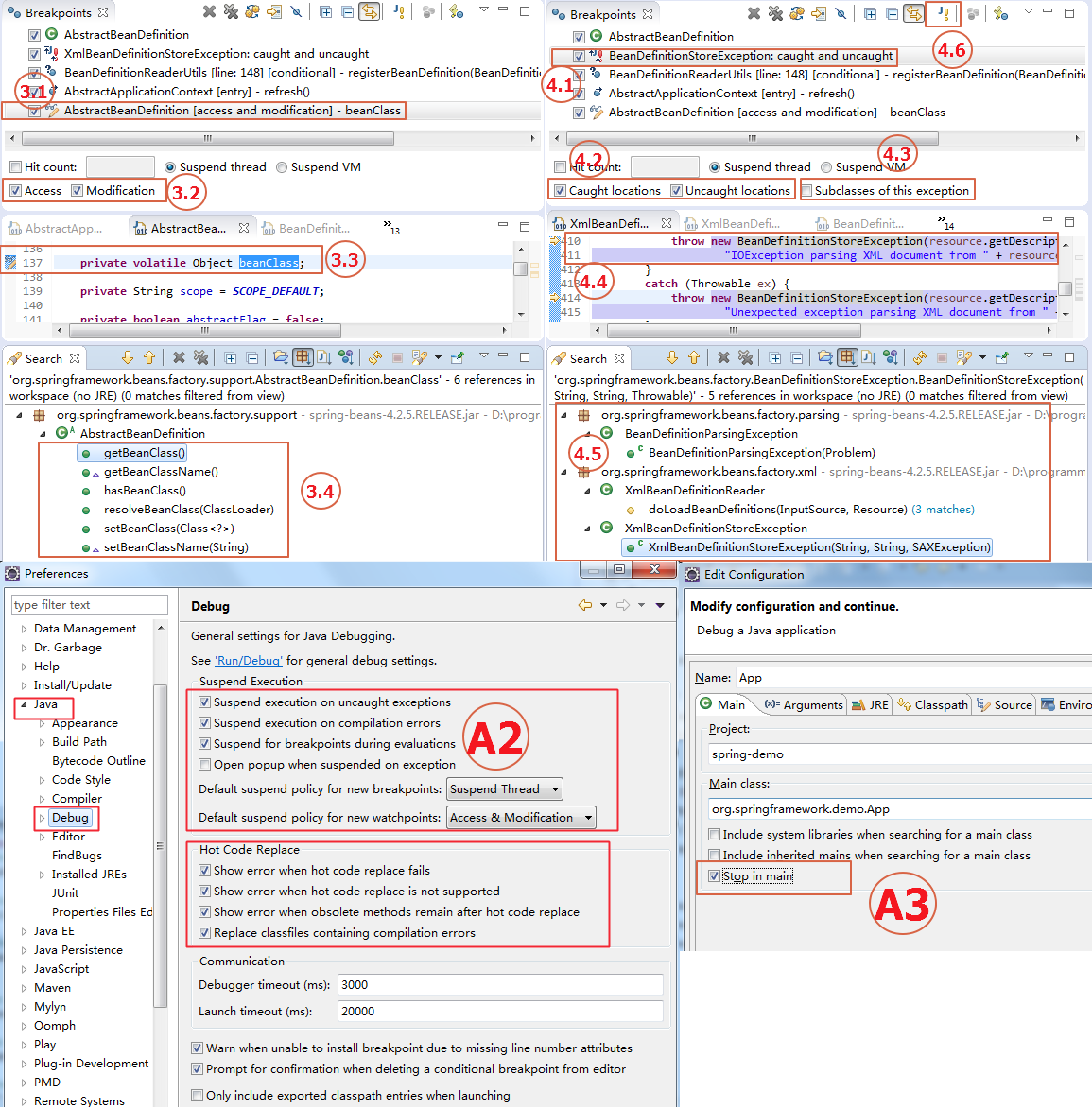
4、异常断点： 异常断点可以通过4.6添加，或者点击日志信息中输出的异常类信息添加。   
异常断点4.1，系统发生异常时，在被捕获异常的抛出位置处或者程序未捕获的异常抛出处4.2、4.4, 挂起线程/VM， 也可以指定是否包括异常的子类也被检测4.3、4.5。

另外除了以上正常设置的异常挂起，从java->debug中可以设置挂起执行，主要有下面两个：

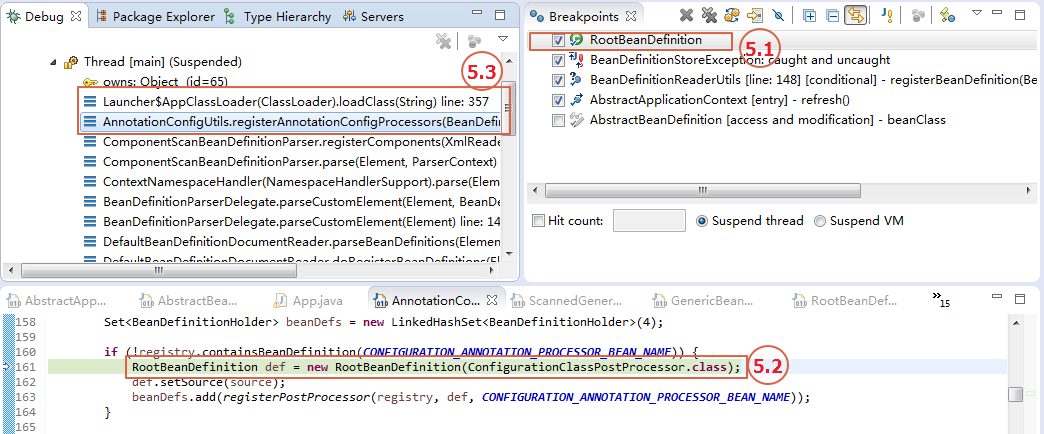
1、是否在发生全局未捕获时挂起(suspend execution on uncaught exceptions)，调试时，老是有异常挂起影响调试，但是没有设置异常断点的情况，就可以勾选掉这个选项；

2、是否在编译错误时挂起，一般在边调试边改代码时会发生这种情况；

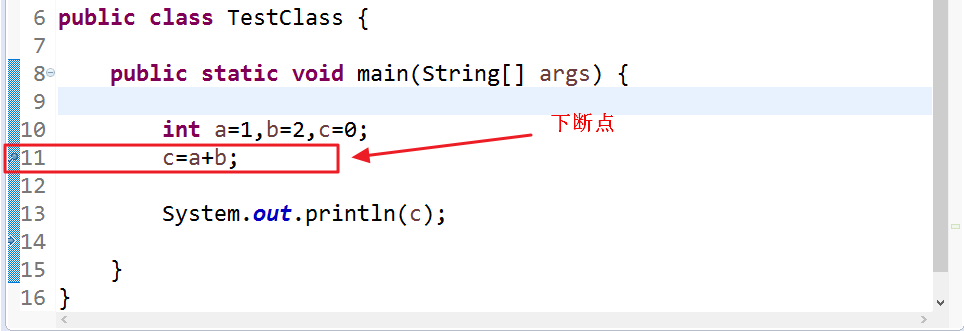
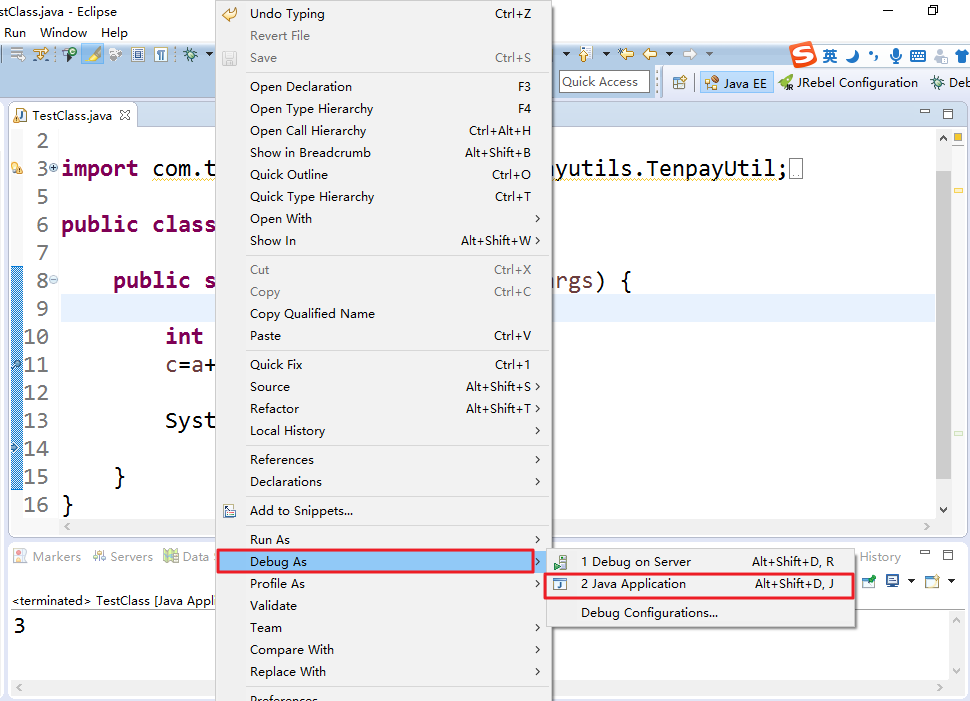
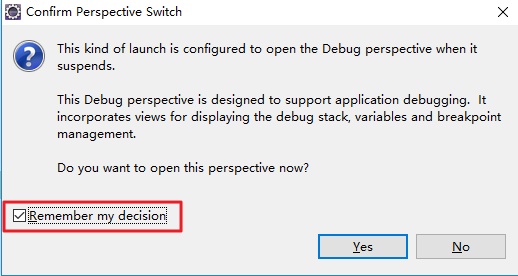
另外要提一个的是有main方法启动的应用，可以在调试配置中勾选stop in main A3, 程序进入时，会挂起线程，等待调试。



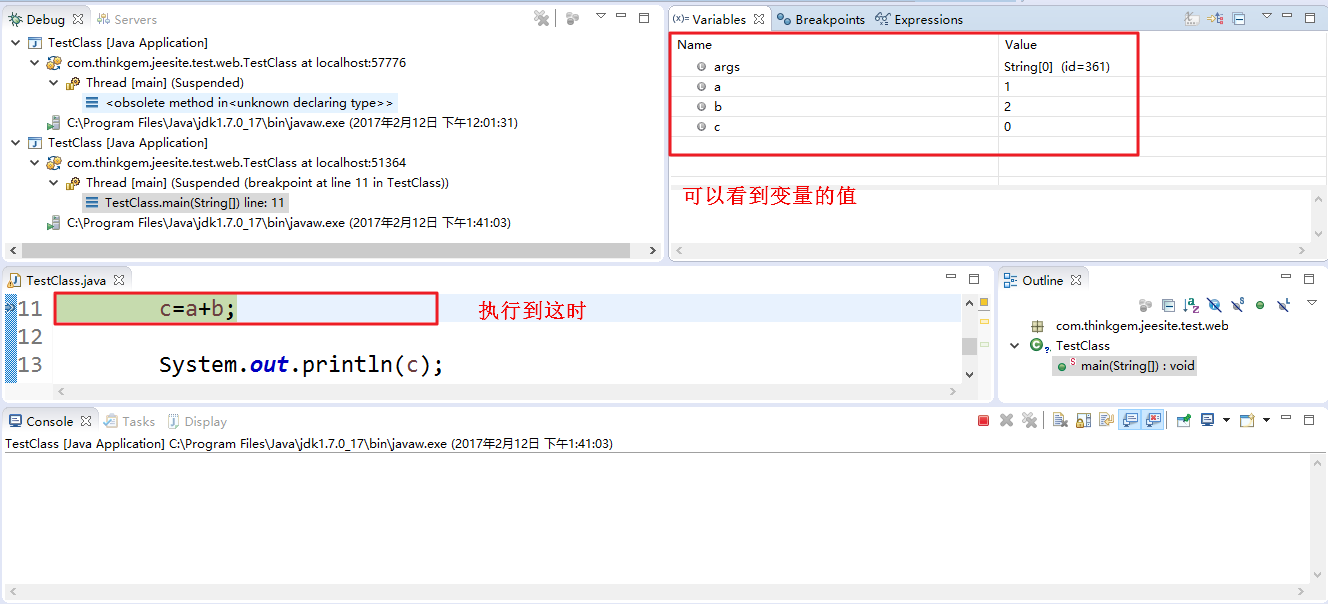
5、类加载断点: 在类名上打的断点5.1。接口上是打不了类加载断点的，但是抽象类是可以的，只是在调试的时候，断点不会明显进入classloader中，单步进入知会进入到子类的构造方法中，非抽象类在挂起线程后单步进入就会到classloader中（如果没有filter过滤掉的话）5.3。类加载断点不管是打在抽象或者非抽象类上，都会在类第一次加载或者第一个子类第一次被加载时，挂起线程/VM5.2。



**2.调试程序  
  
  
1、调试本地 Java 语言程序**  
  
在所有调试中，调试一个Java程序是最简单的，主要有设置断点、启动调试、单步执行、结束调试几步。  
  
1）设置断点：

  
  
2）启动调试：Eclipse提供四种方式来启动程序(Launch)的调试，分别是通过菜单(Run –> Debug)、图标(“绿色臭虫”)、右键->Debug As以及快捷键(F11)，在这一点上，与其他命令(例如Run)类似。  
  
  
  
  
弹出提示，需要切换到调试（Debug）工作区，勾选“Remember my decision”，记住选择，则下次不再提示，然后点击【Yes】。  
  


3）单步执行：主要使用前面讲过的几个视图进行调试，其中debug视图中的几个按钮有快捷键:  
  
Step Retuen(F7)  
  
Step Over (F6)  
  
Step Into (F5)



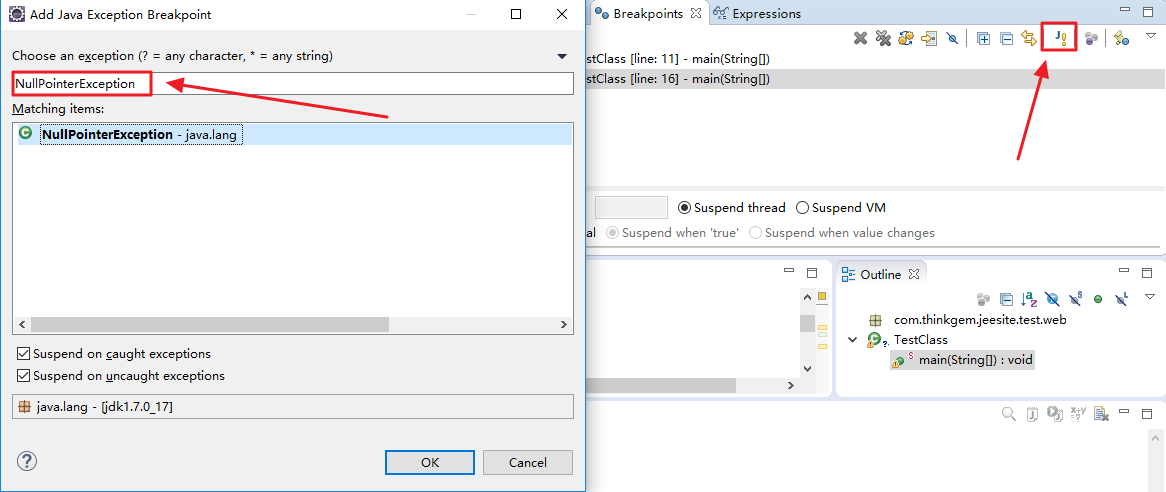
结束调试：通过Terminate命令终止对本地程序的调试。

**二、案例**

**场景一：小明写了一个任务执行者，该执行者不间断的执行一些任务，在现网上运行了一段时间后，发现有概率的出现一些故障，发现运行一段时间后，该任务者异常退出了，退出的因为是空指针，可以小明想要在本地debug，不知道断点打在哪里，该问题是概率事件，不一定会出现，所以小明debug几遍下来后，头晕眼花，连代码都看不清楚了，小明想要是能有个断点每当出现空指针异常的时候就停下来让他发现问题，那该多好呀。**

**异常断点**

异常断点：在定位问题的过程中，常常会遇到断点无法打到合适的位置，以至于和问题的触发点千差万别，所以这个时候不妨试试异常断点，顾名思义，异常断点是指抛出某种异常后自动挂起的断点。  
  
点击红色部位，增加一个异常断点

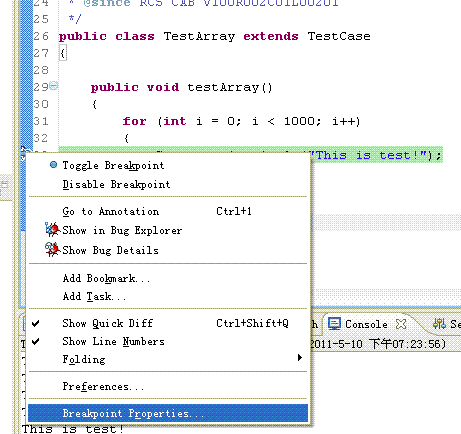


输入想要定位的异常类型，例如NullPointerException，这样系统中抛出任何NullPointerException异常后，都会挂起当前线程，给你机会去定位问题。

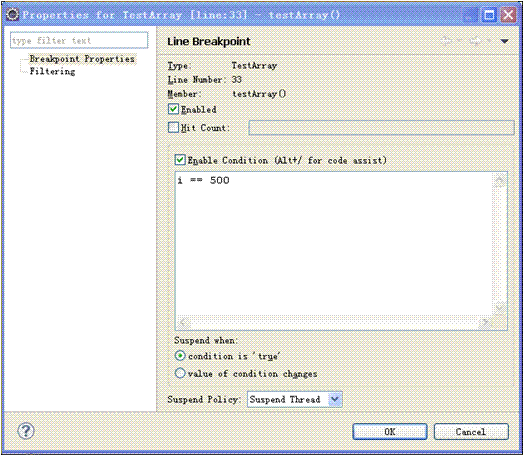
**场景二：小明写了一个巨大的循环，在调测代码时，小明发现每当循环到第100000次的时候，就是出现问题，没有达到自己的预期，于是小明在循环里打了个断点，想看看到底怎么回事，可小明万万没有想到，想要到达100000次循环是多么的困难，小明这个时候已经开始浮想联翩，如果能有这样的断点：**  
  
             If 循环次数== 100000，线程停下来

条件断点

如右图，循环1000次，如果想要在循环到500次的时候停下来，可以创建一个条件断点，右击断点悬着Breakpoint Properties。



选中Enable Condition  
  
在空白处，添加你自己的条件，如果条件返回true，线程会被挂起，如果为false，则忽略该异常  
  
Hit Count为该断点经过多少次后，正式挂起线程，如果设置为500，则表达前499次，经过该断点都不会停下，当第500次，该断点会挂起当前线程。



表达式

表达式可以查看一些在当前代码中没有的命令行，方便定位问题。

**场景三：小明最近遇到一个难题，在调用一个第三方插件时总是会有问题，小明怀疑是第三方插件的bug，但小明没有找到源码不能进行debug，小明该怎么办呢？**  
  
  
Debug定位第三方插件的问题

1.使用反编译工具将代码反编译  
  
2.将反编译后的源码进行过滤  
  
3.修复源码编译错误  
  
4.进行debug