修复 Java 内存模型

**简介：** 活跃了将近三年的 JSR 133，近期发布了关于如何修复 Java 内存模型（Java Memory Model, JMM）的公开建议。原始 JMM 中有几个严重缺陷，这导致了一些难度高得惊人的概念语义，这些概念原来被认为很简单，如 volatile、final 以及 synchronized。在这一期的 Java 理论与实践 中，Brian Goetz 展示了如何加强 volatile 和 final 的语义，以修复 JMM。这些更改有些已经集成在 JDK 1.4 中

Java 平台把线程和多处理技术集成到了语言中，这种集成程度比以前的大多数编程语言都要强很多。该语言对于平台独立的并发及多线程技术的支持是野心勃勃并且是具有开拓性的，或许并不奇怪，这个问题要比 Java 体系结构设计者的原始构想要稍微困难些。关于同步和线程安全的许多底层混淆是 Java 内存模型 (JMM)的一些难以直觉到的细微差别，这些差别最初是在 Java Language Specification 的第 17 章中指定的，并且由 JSR 133 重新指定。

例如，并不是所有的多处理器系统都表现出 *缓存一致性（cache coherency）*；假如有一个处理器有一个更新了的变量值位于其缓存中，但还没有被存入主存，这样别的处理器就可能会看不到这个更新的值。在缓存缺乏一致性的情况下，两个不同的处理器可以看到在内存中同一位置处有两种不同的值。这听起来不太可能，但是这却是故意的 —— 这是一种获得较高的性能和可伸缩性的方法 —— 但是这加重了开发者和编译器为解决这些问题而编写代码的负担。

**什么是内存模型，我为什么需要一个内存模型？**

内存模型描述的是程序中各变量（实例域、静态域和数组元素）之间的关系，以及在实际计算机系统中将变量存储到内存和从内存取出变量这样的低层细节。对象最终存储在内存中，但编译器、运行库、处理器或缓存可以有特权定时地在变量的指定内存位置存入或取出变量值。例如，编译器为了优化一个循环索引变量，可能会选择把它存储到一个寄存器中，或者缓存会延迟到一个更适合的时间，才把一个新的变量值存入主存。所有的这些优化是为了帮助实现更高的性能，通常这对于用户来说是透明的，但是对多处理系统来说，这些复杂的事情可能有时会完全显现出来。

JMM 允许编译器和缓存以数据在处理器特定的缓存（或寄存器）和主存之间移动的次序拥有重要的特权，除非程序员已经使用 synchronized 或 final 明确地请求了某些可见性保证。这意味着在缺乏同步的情况下，从不同的线程角度来看，内存的操作是以不同的次序发生的。

与之相对应地，像 C 和 C++ 这些语言就没有显示的内存模型 —— 但 C 语言程序继承了执行程序处理器的内存模型（尽管一个给定体系结构的编译器可能知道有关底层处理器的内存模型的一些情况，并且保持一致性的一部分责任也落到了该编译器的头上）。这意味着并发的 C 语言程序可以在一个，而不能在另一个，处理器体系结构上正确地运行。虽然一开始 JMM 会有些混乱，但这有个很大的好处 —— 根据 JMM 而被正确同步的程序能正确地运行在任何支持 Java 的平台上。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#ibm-pcon)

**原始 JMM 的缺点**

虽然在 *Java Language Specification* 的第 17 章指定的 JMM 是一个野心勃勃的尝试，它尝试定义一个一致的、跨平台的内存模型，但它有一些细微而重要的缺点。 synchronized 和 volatile 的语义很让人混淆，以致于许多有见地的开发者有时选择忽略这些规则，因为在旧的存储模型下编写正确同步的代码非常困难。

旧的 JMM 允许一些奇怪而混乱的事情发生，比如 final 字段看起来没有那种设置在构造函数里的值（这样使得想像上的不可变对象并不是不可变的）和内存操作重新排序的意外结果。这也防止了其他一些有效的编译器优化。如果您阅读了关于双重检查锁定问题（double-checked locking problem）的任何文章（参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#resources)），您将会记得内存操作重新排序是多么的混乱，以及当您没有正确地同步（或者没有积极地试图避免同步）时，细微却严重的问题会如何暗藏在您的代码中。更糟糕的是，许多没有正确同步的程序在某些情况下似乎工作得很好，例如在轻微的负载下、在单处理器系统上，或者在具有比 JMM 所要求的更强的内存模型的处理器上。

“重新排序”这个术语用于描述几种对内存操作的真实明显的重新排序的类型：

* 当编译器不会改变程序的语义时，作为一种优化它可以随意地重新排序某些指令。
* 在某些情况下，可以允许处理器以颠倒的次序执行一些操作。
* 通常允许缓存以与程序写入变量时所不相同的次序把变量存入主存。

从另一线程的角度来看，任何这些条件都会引发一些操作以不同于程序指定的次序发生 —— 并且忽略重新排序的源代码时，内存模型认为所有这些条件都是同等的。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#ibm-pcon)

**JSR 133 的目标**

JSR 133 被授权来修复 JMM，它有几个目标：

* 保留现有的安全保证，包括类型安全。
* 提供 *无中生有安全性（out-of-thin-air safety）*。这意味着变量值并不是“无中生有”地创建的 —— 所以对于一个线程来说，要观察到一个变量具有变量值 X，必须有某个线程以前已经真正把变量值 X 写入了那个变量。
* “正确同步的”程序的语义应该尽可能简单直观。这样，“正确同步的”应该被正式而直观地定义（这两种定义应该相互一致）。
* 程序员应该要有信心创建多线程程序。当然，我们没有魔法使得编写并发程序变得很容易，但是我们的目标是为了减轻程序员理解内存模型所有细节的负担。
* 跨大范围的流行硬件体系结构上的高性能 JVM 实现应该是可能的。现代的处理器在它们的内存模型上有着很大的不同；JMM 应该能够适合于实际的尽可能多的体系结构，而不会以牺牲性能为代价。
* 提供一个同步习惯用法（idiom），以允许我们发布一个对象并且使得它不用同步就可见。这是一种叫做 *初始化安全（initialization safety）*的新的安全保证。
* 对现有代码应该只有最小限度的影响。

值得注意的是，有漏洞的技术（如双重检查锁定）在新的内存模型下仍然有漏洞，并且“修复”双重检查锁定技术并不是新内存模型所致力的一个目标。（但是， volatile 的新语义允许通常所提出的其中一个双重检查锁定的可选方法正确地工作，尽管我们不鼓励这种技术。）

从 JSR 133 process 变得活跃的三年来，人们发现这些问题比他们认为重要的任何问题都要微妙得多。这就是作为一个开拓者的代价！最终正式的语义比原来所预料的要复杂得多，实际上它采用了一种与原先预想的完全不同的形式，但非正式的语义是清晰直观的，将在本文的第 2 部分概要地说明。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#ibm-pcon)

**同步和可见性**

大多数程序员都知道， synchronized 关键字强制实施一个互斥锁（互相排斥），这个互斥锁防止每次有多个线程进入一个给定监控器所保护的同步语句块。但是同步还有另一个方面：正如 JMM 所指定，它强制实施某些内存可见性规则。它确保了当存在一个同步块时缓存被更新，当输入一个同步块时缓存失效。因此，在一个由给定监控器保护的同步块期间，一个线程所写入的值对于其余所有的执行由同一监控器所保护的同步块的线程来说是可见的。它也确保了编译器不会把指令从一个同步块的内部移到外部（虽然在某些情况下它会把指令从同步块的外部移到内部）。JMM 在缺乏同步的情况下不会做这种保证 —— 这就是只要有多个线程访问相同的变量时必须使用同步（或者它的同胞，易失性）的原因。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#ibm-pcon)

**问题 1：不可变对象不是不可变的**

JMM 的其中一个最惊人的缺点是，不可变对象似乎可以改变它们的值（这种对象的不变性旨在通过使用 final 关键字来得到保证）。（Public Service Reminder：让一个对象的所有字段都为 final 并不一定使得这个对象不可变 —— 所有的字段 *还* 必须是原语类型或是对不可变对象的引用。）不可变对象（如 String ）被认为不要求同步。但是，因为在将内存写方面的更改从一个线程传播到另一个线程时存在潜在的延迟，所以有可能存在一种竞态条件，即允许一个线程首先看到不可变对象的一个值，一段时间之后看到的是一个不同的值。

这是怎么发生的呢？考虑到 Sun 1.4 JDK 中 String 的实现，这儿基本上有三个重要的决定性字段：对字符数组的引用、长度和描述字符串开始的字符数组的偏移量。 String 是以这种方式实现的，而不是只有字符数组，因此字符数组可以在多个 String 和 StringBuffer 对象之间共享，而不需要在每次创建一个 String 时都将文本拷贝到一个新的数组里。例如，String.substring() 创建了一个可以与原始的 String 共享同一个字符数组的新字符串，并且这两个字符串仅仅只是在长度和偏移量上有所不同。

假设您执行以下的代码：

|  |
| --- |
| String s1 = "/usr/tmp";  String s2 = s1.substring(4); // contains "/tmp" |

字符串 s2 将具有大小为 4 的长度和偏移量，但是它将同 s1 共享包含“ /usr /tmp ”的同一字符数组。在 String 构造函数运行之前， Object 的构造函数将用它们默认的值初始化所有字段，包括决定性的长度和偏移字段。当 String 构造器运行时，字符串长度和偏移量被设置成所需要的值。但是在旧的内存模型下，在缺乏同步的情况下，有可能另一个线程会临时地看到偏移量字段具有初默认值 0，而后又看到正确的值 4。结果是 s2 的值从“ /usr ”变成了“ /tmp ”。这并不是我们所想要的，而且在所有 JVM 或平台这是不可能的，但是旧的内存模型规范允许这样做。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#ibm-pcon)

**问题 2：重新排序易失性和非易失性存储**

另一个主要领域是与 volatile 字段的内存操作重新排序有关，这个领域中现有 JMM 引起了一些非常混乱的结果。现有 JMM 表明易失性的读和写是直接和主存打交道的，这样避免了把值存储到寄存器或者绕过处理器特定的缓存。这使得多个线程一般能看见一个给定变量的最新的值。可是，结果是这种 volatile 定义并没有最初所想像的那样有用，并且它导致了 volatile 实际意义上的重大混乱。

为了在缺乏同步的情况下提供较好的性能，编译器、运行库和缓存通常被允许重新排序普通的内存操作，只要当前执行的线程分辨不出它们的区别。（这就是所谓的 *线程内似乎是串行的语义（within-thread as-if-serial semantics）*。）但是，易失性的读和写是完全跨线程安排的，编译器或缓存不能在彼此之间重新排序易失性的读和写。遗憾的是，通过参考普通变量的读和写，JMM 允许易失性的读和写被重新排序，这意味着我们不能使用易失性标志作为操作已完成的指示。考虑下面的代码，其意图是假定易失性字段 initialized 用于表明初始化已经完成了。

**清单 1. 使用一个易失性字段作为一个“守卫”变量**

|  |
| --- |
| Map configOptions;  char[] configText;  volatile boolean initialized = false;  . .  // In thread A    configOptions = new HashMap();  configText = readConfigFile(fileName);  processConfigOptions(configText, configOptions);  initialized = true;  . .  // In thread B    while (!initialized)  sleep();  // use configOptions |

这里的思想是使用易失性变量 initialized 担任守卫来表明一套别的操作已经完成了。这是一个很好的思想，但是它不能在旧的 JMM 下工作，因为旧的 JMM 允许非易失性的写（比如写到 configOptions 字段，以及写到由 configOptions 引用Map 的字段中）与易失性的写一起重新排序，因此另一个线程可能会看到 initialized 为 true，但是对于 configOptions 字段或它所引用的对象还没有一个一致的或者说当前的视图。 volatile 的旧语义只承诺正在读和写的变量的可见性，而不承诺其他的变量。虽然这种方法更容易有效地实现，但结果是没有原来所想的那么有用。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02244/#ibm-pcon)

**结束语**

正如 Java Language Specification 第 17 章中所指定的，JMM 有一些严重的缺点，即允许一些看起来合理的程序发生一些非直观的或不合需要的事情。如果正确地编写并发的类太困难的话，那么我们可以说许多并发的类不能按预期工作，并且这是平台中的一个缺点。幸运的是，我们可以在不破坏在旧的内存模型下正确同步的任何代码的同时，创建一个与大多数开发者的直觉更加一致的内存模型，并且这一切已经由 JSR 133 process 完成。下个月，我们将介绍新的内存模型（它的大部分功能已集成到 1.4 JDK 中）的详细信息。

开始编写并发代码是一件困难的事情，语言不应当增加它的难度。虽然 Java 平台从一开始就包括了对线程的支持，包括一个计划为正确同步的程序提供“一次编写，到处运行”保证的、跨平台的内存模型，但是原来的内存模型有一些漏洞。虽然许多 Java 平台提供了比 JMM 所要求的更强的保证，但是 JMM 中的漏洞使得无法容易地编写可以在任何平台上运行的并发 Java 程序。所以在 2001 年 5 月，成立了以修复 Java 内存模型为目的的 JSR 133。 [上个月](http://www.ibm.com/developerworks/cn/java/j-jtp02244/)，我讨论了其中一些漏洞，这个月，我们将讨论如何堵住它们。

**修复后的可见性**

理解 JMM 所需要的一个关键概念是 可见性（visibility）——如何知道当线程 A 执行 someVariable?=?3 时，其他线程是否可以看到线程 A 所写的值 3？有一些原因使其他线程不能立即看到 someVariable 的值 3：可能是因为编译器为了执行效率更高而重新排序了指令，也可能是 someVariable 缓存在寄存器中，或者它的值写到写处理器的缓存中、但是还没有刷新到主存中，或者在读处理器的缓存中有一个老的（或者无效的）值。内存模型决定什么时候一个线程可以可靠地“看到”由其他线程对变量的写入。特别是，内存模型定义了保证内存操作跨线程的可见性的 volatile 、 synchronized 和 final 的语义。

## 不要错过本系列的其他内容

第 1 部分“ [什么是 Java 内存模型，最初它是怎样被破坏的？](http://www.ibm.com/developerworks/cn/java/j-jtp02244/)”（2004 年 2 月）

当线程为释放相关监视器而退出一个同步块时，JMM 要求本地处理器缓冲刷新到主存中。（实际上，内存模型不涉及缓存——它涉及一个抽象（ 本地内存）, 它包围了缓存、注册表和其他硬件和编译优化。）与此类似，作为获得监视的一部分，当进入一个同步块时，本地缓存失效，使之后的读操作直接进入主内存而不是本地缓存。这一过程保证当变量是由一个线程在由给定监视器保护的同步块中写入，并由另一个线程在由同一监视器保护的同步块中读取时，对变量的写可被读线程看到。如果没有同步，则 JMM 不提供这种保证——这就是为什么在多个线程访问同一个变量时，必须使用同步（或者它的更年轻的同胞 volatile ）。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp03304/#ibm-pcon)

**对 volatile 的新保证**

volatile 原来的语义只保证 volatile 字段的读写直接在主存而不是寄存器或者本地处理器缓存中进行，并且代表线程对 volatile 变量进行的这些操作是按线程要求的顺序进行的。换句话说，这意味着老的内存模型只保证正在读或写的变量的可见性，不保证写入其他变量的可见性。虽然可以容易实现它，但是它没有像最初设想的那么有用。

虽然对 volatile 变量的读和写不能与对其他 volatile 变量的读和写一起重新排序，但是它们仍然可以与对 nonvolatile 变量的读写一起重新排序。在 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-jtp02244/) 中，介绍了清单 1 的代码（在旧的内存模型中）是如何不足以保证线程 B 看到configOptions 及通过 configOptions 间接可及的所有变量（如 Map 元素）的正确值，因为 configOptions 的初始化可能已经随 volatile initialized 变量进行重新排序。

**清单 1. 用一个 volatile 变量作为“守护”**

|  |
| --- |
| Map configOptions;  char[] configText;  volatile boolean initialized = false;  // In Thread A  configOptions = new HashMap();  configText = readConfigFile(fileName);  processConfigOptions(configText, configOptions);  initialized = true;  // In Thread B  while (!initialized)  sleep();  // use configOptions |

不幸地，这是 volatile 常见用例——用一个 volatile 字段作为“守护”表明已经初始化了一组共享变量。JSR 133 Expert Group 决定让 volatile 读写不能与其他内存操作一起重新排序是有意义的——可以准确地支持这种和其他类似的用例。在新的内存模型下，如果当线程 A 写入 volatile 变量 V 而线程 B 读取 V 时，那么在写入 V 时，A 可见的所有变量值现在都可以保证对 B 是可见的。结果就是作用更大的 volatile 语义，代价是访问 volatile 字段时会对性能产生更大的影响。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp03304/#ibm-pcon)

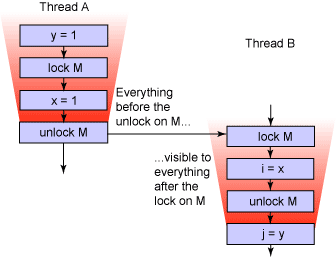
**在这之前发生了什么？**

像对变量的读写这样的操作，在线程中是根据所谓的“程序顺序”——程序的语义声明它们应当发生的顺序——排序的。（编译器实际上对在线程中使用程序顺序是可以有一些自由的——只要保留了 as-if-serial 语义。）在不同线程中的操作完全不一定要彼此排序——如果启动两个线程并且它们对任何公共监视器都不用同步执行、或者涉及任何公共 volatile 变量，则完全 无法准确地预言一个线程中的操作（或者对第三个线程可见）相对于另一个线程中操作的顺序。

此外，排序保证是在线程启动、一个线程参与另一个线程、一个线程获得或者释放一个监视器（进入或者退出一个同步块）、或者一个线程访问一个 volatile 变量时创建的。JMM 描述了程序使用同步或者 volatile 变量以协调多个线程中的活动时所进行的的顺序保证。新的 JMM 非正式地定义了一个名为 happens-before 的排序，它是程序中所有操作的部分顺序，如下所示：

* 线程中的每一个操作 happens-before这个线程中在程序顺序中后面出现的每一个操作
* 对监视器的解锁 happens-before同一监视器上的所有后续锁定
* 对 volatile 字段的写 happens-before同一 volatile 的每一个后续读
* 对一个线程的 Thread.start() 调用 happens-before在启动的线程中的所有操作
* 线程中的所有操作 happens-before 从这个线程的 Thread.join() 成功返回的所有其他线程

这些规则中的第三个——控制对 volatile 变量的读写，是新的并且修正了清单 1 中的例子的问题。因为对 volatile 变量initialized 的写是在初始化 configOptions 之后发生的， configOptions 的使用是在 initialized 的读后发生的，而对initialized 的读是在对 initialized 的写后发生的，因此可以得出结论，线程 A 对 configOptions 的初始化是在线程 B 使用 configOptions 之前发生的。因而 configOptions 和通过它可及的变量对于线程 B 是可见的。

**图 1. 用同步保证跨线程的内存写的可见性**  
 

**数据争用**

当有一个变量被多个线程读、被至少一个线程写、并且读和写不是按 hanppens-before 关系排序的时，程序就称为有 数据争取（data race），因而不是一个“正确同步”的程序。

**这是否修改了双重检查锁定的问题？**

对双重检查锁定问题提出的一种修复是使包含迟缓初始化的实例的字段为一个 volatile 字段。（有关双重检查锁定的问题和对为什么所建议的算法修复不能解决问题的说明请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp03304/#resources)。）在旧的内存模型中，这不能使双重检查锁定成为线程安全的，因为对 volatile 字段的写仍然会与对其他 nonvolatile 字段的写（如新构造的对象的字段）一起重新排序，因而 volatile 实例引用仍然可能包含对一个未构造完的对象的引用。

在新的内存模型中，对双重检查锁定的这个“修复”使 idiom 线程安全。但是仍然不意味着应当使用这个 idiom！双重检查锁定的要点是，它假定是性能优化的，设计用于消除公共代码路径的同步，很大程度上因为对于早期的 JDK 来说，同步是相对昂贵的。不仅非竞争的同步已经便宜 多 了，而且对 volatile 语义的新改变也使它在某些平台上比旧的语义昂贵得多。（实际上，对 volatile 字段的每一次读或者写都像是“半个”同步——对 volatile 的读有与监视器所获得的同样的内存语义，对 volatile 的写有与监视器所释放的同样的语义。）所以如果双重检查锁定的目标是提供比更直观的同步方式更好的性能，那么这个“修复的”版本也没有多大帮助。

不使用双重检查锁定，而使用 Initialize-on-demand Holder Class idiom，它提供了迟缓初始化，是线程安全的，而且比双重检查锁定更快且没那么混乱：

**清单 2. Initialize-On-Demand Holder Class idiom**

|  |
| --- |
| private static class LazySomethingHolder {  public static Something something = new Something();  }  ...  public static Something getInstance() {  return LazySomethingHolder.something;  } |

这个 idiom 由属于类初始化的操作（如静态初始化器）保证对使用这个类的所有线程都是可见的这一事实衍生其线程安全性，内部类直到有线程引用其字段或者方法时才装载这一事实衍生出迟缓初始化。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp03304/#ibm-pcon)

**初始化安全性**

新的 JMM 还寻求提供一种新的 初始化安全性 保证——只要对象是正确构造的（意即不会在构造函数完成之前发布对这个对象的引用），然后所有线程都会看到在构造函数中设置的 final 字段的值，不管是否使用同步在线程之间传递这个引用。而且，所有可以通过正确构造的对象的 final 字段可及的变量，如用一个 final 字段引用的对象的 final 字段，也保证对其他线程是可见的。这意味着如果 final 字段包含，比如说对一个 LinkedList 的引用，除了引用的正确的值对于其他线程是可见的外，这个 LinkedList 在构造时的内容在不同步的情况下，对于其他线程也是可见的。结果是显著增强了 final的意义——可以不用同步安全地访问这个 final 字段，编译器可以假定 final 字段将不会改变，因而可以优化多次提取。

**Final 意味着最终**

在 [第 1 部分](http://www.ibm.com/developerworks/cn/java/j-jtp02244/)描述了在旧的内存模型中，final 字段的值似乎可以改变的一种机制——在不使用同步时,另一个线程会首先看到 final 字段的默认值，然后看到正确的值。

在新的内存模型中，在构造函数的 final 字段的写与在另一个线程中对这个对象的共享引用的初次装载之间有一个类似于 happens-before 的关系。当构造函数完成任务时，对 final 字段的所有写（以及通过这些 final 字段间接可及的变量）变为“冻结”，所有在冻结之后获得对这个对象的引用的线程都会保证看到所有冻结字段的冻结值。初始化 final 字段的写将不会与构造函数关联的冻结后面的操作一起重新排序。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp03304/#ibm-pcon)

**结束语**

JSR 133 显著增强了 volatile 的语义，这样就可以可靠地使用 volatile 标志表明程序状态被另一个线程改变了。作为使 volatile 更“重量级”的结果，使用 volatile 的性能成本更接近于某些情况下同步的性能成本，但是在大多数平台上性能成本仍然相当低。JSR 133 还显著地增强了 final 的语义。如果一个对象的引用在构造阶段不允许逸出（escape），那么一旦构造函数完成，并且线程发布了对另一个对象的引用，那么在不使用同步的条件下，这个对象的 final 字段就保证对所有其他线程是可见的、正确的并且是不变的。

这些改变极大地加强了并发程序中不变对象的效用，不变对象最终成为固有的线程安全（就像它们所要成为的那样），即使使用数据争用在线程之间将引用传递给不变对象。

初始化安全性的一个告诫是对象的引用不许“逸出”其构造函数——构造函数不应直接或者间接发布对正在构造的对象的引用。这包括发布对 nonstatic 内部类的引用，并一般要避免在构造函数中启动线程。有关安全构造的更详细说明，请参阅[参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp03304/#resources)。