# 作者

## 第一版的作者

Greg Bollella是Sun Microsystems实验室的一位杰出工程师和实时Java的首席研究员，他是RTSJ的创始规范负责人，同时也是IBM的高级架构师。他拥有北卡罗来纳大学教堂山分校(University of North Carolina at Chapel Hill)计算机科学博士学位。论文研究方向为实时调度理论和实时系统实现。

Ben Brosgol是Ada Core Technologies, Inc.的高级技术人员。他长期参与编程语言的设计和实现，专注于Ada和实时支持，并自1997年以来一直提供java相关服务。Ben持有哈佛大学(Harvard University)应用数学博士学位和阿默斯特学院(Amherst College)学士学位。

Peter Dibble是微软系统公司的资深科学家，从RTSJ的最初设计到RTSJ的1.0版本的最终发布。在微软，他为实时系统设计、编码和分析系统软件超过12年，作为微软Java团队的一员，Peter从1997年初就开始参与Java虚拟机。他拥有罗切斯特大学计算机科学博士和硕士学位。

Steve Furr目前在QNX软件系统工作，在那里他负责QNX中微子操作系统的Java技术。他毕业于西蒙弗雷泽大学，获得计算机科学学士学位。

James Gosling是Sun Microsystems的研究员，他是Java编程语言的创始人。他的编程生涯始于为科学仪器开发实时软件。他拥有卡内基梅隆大学(Carnegie-Mellon University)计算机科学博士学位和硕士学位，以及卡尔加里大学(University of Calgary)计算机科学学士学位。

David Hardin, aJile Systems的首席技术官和联合创始人，曾在Rockwell Collins从事安全关键型计算机系统架构、形式化方法和定制微处理器设计工作，并被任命为1997年Rockwell年度工程师。他拥有堪萨斯州立大学电子与计算机工程博士学位。

Mark Turnbull自1983年以来一直是Nortel Networks的员工。他的大部分经验都是在专有语言设计、编译器设计和实时系统领域。

鲁迪·贝利亚迪(Rudy bellardi)是施耐德自动化高级技术咨询工程师。他为工厂自动化和医疗领域设计并实现了系统、语言和协议。他拥有热那亚大学(University of Genova)电子工程博士学位和乔治华盛顿大学(George Washington University)计算机科学应用科学家专业学位。

## 第二版的作者

该规范版本的大部分工作由RTSJ技术解释委员会的活跃成员分担。修改是与彼得·迪布尔协商，然后由整个小组检查和批准。尽管每个作者都对某些章节负有特定的责任，但整个规范是一个共同的工作。这里列出了主要贡献者。

鲁迪·贝利亚迪(Rudy bellardi)是施耐德自动化高级技术咨询工程师。他为工厂自动化和医疗领域设计并实现了系统、语言和协议。他积极参与工业实时通信工作。他拥有热那亚大学(University of Genova)电子工程博士学位和乔治华盛顿大学(George Washington University)计算机科学应用科学家专业学位。他是时间、时钟和计时器章节的主要贡献者。

Ben Brosgol是Ada Core Technologies, Inc.的高级技术人员。他长期参与编程语言的设计和实现，专注于Ada和实时支持，并自1997年以来一直提供java相关服务。Ben持有哈佛大学(Harvard University)应用数学博士学位和阿默斯特学院(Amherst College)学士学位。他是同步章节的主要贡献者。

Peter Dibble是TimeSys的杰出工程师，他是实现并支持RTSJ参考实现、RTSJ技术一致性套件以及RTSJ的第一个商业实现的团队成员。他担任RTSJ规范解释和维护的技术领导，因为TimeSys承担了RTSJ维护领导的角色，并在过程的后期转移到维护领导。他是这个版本的编辑，是内存章节的主要贡献者，也是调度和线程章节的团队成员。

David Holmes是位于澳大利亚布里斯班的DLTeCH Pty Ltd的董事和首席科学家。他在Java技术方面的工作重点是语言和虚拟机中的并发和同步支持，他目前正在开发实时Java虚拟机。David是正在Java Community Process下开发的JSR-166“并发实用程序”专家组的成员，也是《Java编程语言》第三版和第四版的合著者。David于1999年在悉尼麦考瑞大学完成了面向对象系统同步领域的博士学位。他是1.0 RTSJ规范和整个修订过程中一个如此强烈和有用的批评者，以至于其他作者都请他帮助修改。他在各个方面都帮助改进了规范，但在日程安排方面的贡献尤其大。

Andy Wellings是英国约克大学计算机科学系实时系统教授。他的研究兴趣集中在两个相关的计算领域:实时编程语言和操作系统的设计、使用和实现;以及通用分布式操作系统的设计和使用。Wellings教授发表了100多篇技术论文和报告，包括5部教材。他教授操作系统、实时系统、网络和分布式系统等课程。他是异步那一章的主要贡献者，也是调度和线程那一章的团队成员。

## 其他贡献者

无数人为RTSJ做出了贡献，但有几个人投入了很多时间，即使在这个努力工作的群体中，他们也脱颖而出:

Peter Haggar是IBM的高级软件工程师，也是Java社区的杰出成员，他在Greg Bollella跳槽到Sun Microsystems时担任了IBM RTSJ规范负责人的角色。他通过JCP执行委员会的公开审查和批准看到了规范，然后把维护领导的角色交给了Doug Locke。他拥有克拉克森大学的计算机科学学士学位。

C. Douglass Locke，是TimeSys的技术副总裁，在此规范的大部分工作期间是RTSJ的维护主管。他管理参考实施和技术一致性工具包项目的大部分进展。他拥有卡内基梅隆大学(Carnegie-Mellon University)的博士学位，自从计算机从穿孔板开始运行以来，他一直在嵌入式实时领域工作。道格目前是一名独立顾问。

实现参考实现的团队对规范做出了很大的贡献。首先，他们将专家组的梦想与可实现的现实分开。其次，通过提供我们可以使用的实现，他们发现了可用性方面的严重问题。第三，他们成为规范细节方面的专家，并帮助研究出对作用域内存和异步控制传输的最终修改。

三位RI工程师因他们的贡献而脱颖而出:Dionisio deNizVillasenor卡内基梅隆大学计算机科学博士研究生，Scott Robbins卡内基梅隆大学计算机科学学士，Pratik Solanki卡内基梅隆大学计算机科学硕士。

Alden Dima和国家标准与技术研究所多年来一直支持RTSJ邮件列表。这几乎是无形的支持工作，但它很重要，我们感激它。

# 再版前言

在编写于RTSJ完成之前的Java实时规范第一版的前言中，Doug Jensen说他希望RTSJ“成为第一个在商业和技术上都取得成功的实时编程语言”。利用现在由后见之明所给予的主要优势，似乎很明显，他的乐观并非错置，尽管他的预测尚未得到完全证实。

事实上，自2001年RTSJ正式发布以来，RTSJ及其衍生的社区已经稳步、持续地成熟起来，现在已经有了几个可行的实现，许多研究论文，以及许多使用RTSJ的主要实时或嵌入式项目。因此，现在要预测RTSJ将成为第一种在商业和技术上都取得成功的实时编程语言，并不需要太多的预见性。

有趣的是，与许多先前的技术进步一样，RTSJ的适用性领域已经被证明比它的设计者最初设想的要大。显然，RTSJ的目标是实时系统，可以描述为系统的正确性需要推理其时间属性除了其功能属性，如飞行控制或工业传感器控制系统。此外，现在已经很清楚，RTSJ提供的功能极大地增强了Java在嵌入式系统中的使用，而不管它们是否需要实时性能。嵌入式系统可以被描述为其所有者和用户不关心，甚至可能不知道有计算机存在的系统。从手机到空中交通管制系统，这些例子的范围非常广泛。一般来说，是实时系统

因此，现在很清楚，RTSJ的名称不能充分表达其适用范围。RTSJ当然是针对实时系统的，它提供了关键的实时功能，比如对调度的显式支持、有限制的优先级反转、周期性的realtimethread，甚至对硬实时支持的NoHeapRealtimeThreads。然而，对实时性能没有要求或只有最低要求的嵌入式系统的设计者发现RTSJ非常有用，因为它具有AsyncEvents和AsyncEventHandlers、时钟、计时器和POSIXSignalHandlers等功能。事实上，随着实现者越来越多地使用符合RTSJ的平台，他们发现在RTSJ Java中处理POSIX/UNIX/Linux信号比在C或c++中更容易，也更不容易出错。

改变几乎是成功标准的普遍属性。唯一的静态标准是那些没有被使用的标准。因此，RTSJ自最初发布以来一直被不断地澄清、扩展和修正，这一点也不奇怪。它的维护者现在在TimeSys的Peter Dibble的指导下，小心翼翼地确保现有的应用程序和实现不会受到这些更新的影响或最小程度的影响，但这些更新是所有成功标准的一个特征。

RTSJ社区同时也在考虑在RTSJ框架上构建额外的标准。Open Group与一些供应商和开发人员合作，正在考虑基于RTSJ的Java安全关键子集。一些开发团队正在考虑创建RTSJ的概要文件，目标是特定的应用领域，如军事指挥和控制或消费电子产品。

随着RTSJ的不断成熟，其社区有望进一步扩大。在这一点上，看起来RTSJ不仅会成功，而且可以预期它会在实时和嵌入式应用程序的设计方面产生显著的不同，从而在响应性、可维护性、可移植性和可控制的开发成本方面产生重大改进。

事实上，正是这些特性一直是并将继续是所有RTSJ架构师和贡献者的基本目标!

道格洛克

黎巴嫩山,爸爸

# 第一版前言

我希望Java的实时规范能够成为第一个在商业和技术上都取得成功的实时编程语言。

其他编程语言也被用于实时计算领域。然而，就该领域的重大采用而言，没有一项在商业上取得成功。许多是学术研究项目。大多数没有关注管理计算资源以满足应用程序及时性要求的核心实时问题。相反，他们通常强调并行的正交(虽然很重要)主题，以及对嵌入式计算系统(实时计算系统是嵌入式计算系统的子集)的整个领域很重要的其他主题。

Ada 95，包括它的实时系统附件D，在采用和实时技术方面都可能是最成功的实时语言。其中一个原因是，Ada在整个实时计算系统频谱中(在实时语言和操作系统中)非常有效，从传统设备级控制子系统中的小型编程，到企业命令和控制系统中的大型编程。尽管取得了这一成就，但各种非技术因素阻碍了Ada在商业上的成功。

当James Gosling在1995年引入Java编程语言时，基于其最初的目的和设计，它似乎与实时计算领域无关。事实上，它的一些基本原则与实时计算背道而驰。为了促进其操作系统和硬件独立性的主要目标，故意在线程行为、同步、中断、内存管理和输入/输出等方面给该语言提供了较弱的词汇表。然而，这些都属于需要显式管理(通过语言或操作系统)以满足应用程序及时性要求的关键领域。

然而，Java平台“编写一次，随处运行”的承诺，以及Java语言本身作为编程语言的吸引力，在实时(更广泛地说，是嵌入式)领域提供了比在桌面和服务器领域更大的成本节约潜力。台式电脑被“Wintel”双头垄断;服务器只有几种处理器类型和操作系统。实时计算系统有数十种不同的处理器类型和数十种不同的操作系统产品(不包括目前占安装总量一半的定制产品)。POSIX标准没有提供预期的实时应用程序可移植性，因为它允许实现广泛变化的子集。Java平台已经几乎无处不在。实时Java平台必须承诺“仔细编写一次，有条件地在任何地方运行”，这是实现应用程序可重用性的最佳机会。

总的挑战是如何协调Java语言和大多数实时计算的本质差异。必须维护Java的实时规范和Java语言规范的兼容性，同时使前者对于实时计算系统具有成本效益。

大多数参与到，甚至知道实时Java工作的人，包括本书的作者和我，最初都对充分应对这一挑战的可行性非常怀疑。

在Sun的Java社区进程下组建real-time for Java专家组之前，实时Java社区采取了两个重要而不寻常的初始步骤。

第一步是多次召集实时社区的许多代表(在国家标准和技术研究所的主持下)，就Java实时规范的需求达成共识并编写文档。毫不奇怪，当这种共识出现时，它包含了构建小规模、静态、实时子系统的强制性要求，这些子系统是当前使用C和c++的从业者所熟悉的。

更令人惊讶的是，共识还包括了适应高级动态和实时资源管理技术的强制性和可选要求，例如异步控制传输和基于时效性的调度策略，以及构建更大规模的实时系统。这些动态和大型编程实时需求的主要推动力来自已经使用Java语言、或使用Ada语言、或构建防御(主要是命令和控制)系统的社区。

第二步是建立一个公认的实时计算概念和术语词典，以支持关于Java实时规范需求的对话和共识。对于实时社区之外的人来说，实时计算的概念和术语通常没有明确定义(除了大多数实时研究人员)。

要实现Java语言在当前和未来实时计算领域的潜力，下一步是定义和编写Java的实时规范，其中的第一个版本在本书中。理解这个规范还将提高读者对Java语言和实时计算系统的理解。

Greg Bollella是这个规范团队的理想领导者。他招募了一组非常平衡的实时和Java语言专家。他在实际和理论实时计算方面的背景为他温和而坚定地指导团队的丰富而激烈的讨论形成一个连贯的规范做好了准备。

当然，还有更多的工作要做，包括记录用例和例子;执行实现和计算;从已部署的产品中获取经验;以及Java实时规范上的迭代。Java的分布式实时规范也在前面。

实时Java平台不仅准备提供与当前主流实时计算实践和产品相媲美的降低成本的功能，而且还准备在实时计算实践在互联网时代向前发展的过程中发挥领导作用。

e·道格拉斯·詹森

Sherborn,马

# 第一版序言

梦想

1997年，用Java编程语言编写实时应用程序的想法似乎是不现实的。实时程序员谈论的是一致的计时行为而不是绝对的速度，但这并不意味着他们不需要出色的整体性能。Java运行时有时是解释的，而且几乎总是使用垃圾收集器。早期的版本并不以其眩目的表现而闻名。

然而，Java平台已经被整合到实时系统中。构建一个混合系统是相当容易的，它使用C语言来实现具有实时需求的模块和编写到Java平台的其他组件。还可以在硬件上实现Java解释器(为了性能)，并在没有垃圾收集器的情况下集成系统(为了一致的性能)。aJile Systems生成了一个具有可接受的实时特性的Java处理器。

直到1998年夏天，在Java平台上支持实时编程的努力还很分散。来自NewMonics的Kelvin Nilsen和来自国家标准与技术研究所(NIST)的Lisa Carnahan领导了一项工作，来自IBM的Greg Bollella领导了一组在Java技术和实时方面有股份的公司，Sun有一个基于Java平台的内部实时项目。

1998年夏天，三家集团合并。实时需求工作组包括来自NewMonics的Kelvin Nilsen，来自Sun的Bill Foote和Kevin Russell，以及由Greg Bollella领导的公司。它还包括来自实时行业的各种技术人员，以及一些以营销或管理为导向的代表。

需求小组定期召开会议，直到1999年初。它的最终输出是一个文档，名为“Java平台实时扩展的需求”，其中详细描述了小组开发的需求，并给出了这些需求的基本原理。可以在http://www.nist.gov/rt-java网站上找到它。

实现

这个过程中的一个关键事件发生在1998年末，当时Sun创建了Java社区进程。任何认为Java平台需要新功能的人都可以正式请求增强。如果被称为Java规范请求(Java Specification request, JSR)的请求被接受，就会发出对专家的调用。选择规范负责人，然后他或她组成专家组。工作的结果是一个规范、参考实现和测试套件。

在1998年后期，IBM要求Sun接受JSR，即Java的实时规范，这部分是基于需求工作组的工作。Sun以JSR-000001的形式接受了该请求。Greg Bollella被选为规范负责人。他把专家组分成两层。主要组:

格雷格bollella ibm

保罗·鲍曼cyberonics

Ben brosgol aonix /Ada Core Technologies

彼得·迪布尔微软系统公司

史蒂夫furr qnx系统软件实验室

詹姆斯高斯林太阳微系统公司

大卫·哈丁罗克韦尔柯林斯公司/ aJile

马克·特恩布尔北电网络

实际上会编写规范，咨询小组:

鲁迪·贝利亚迪·施耐德自动化公司

奥尔登迪玛nist

道格拉斯·延森·米特雷

亚历山大·卡茨nsicom

三菱电机黑田正弘

洛克·洛克希德·马丁/时代公司

乔治·马列远地点

约翰•mielnik thomson-csf

拉库马Ragunathan卡耐基-梅隆

迈克schuette摩托罗拉

克里斯yurkoski朗讯

西蒙·沃丁顿风河系统

可以作为随时可用的专门知识的汇集，并作为早期草稿的最初审稿人。

这项工作始于1999年3月，顾问和主要小组在芝加哥希尔顿酒店和塔斯酒店举行了一次全体会议。这是一次教育会议，顾问们每个人都提出了一般实时智慧的选择，以及他们所在的实时世界的具体要求。

该规范的基础是在第一次主要小组会议上制定的。它发生在美国少数几个不允许数字或模拟手机通信的文明地区之一，加利福尼亚州的门多西诺。在主要群体的专家看来，这里也是生产世界上最浓芝士披萨的餐厅的所在地。

在一九九九年期间，初级小组每月开会略多于一次，而联合初级小组和顾问小组每月开会略少于一次。我们努力工作，玩得很开心。主要来说，乐趣是与一个由不同的、有才华的软件架构师组成的团队一起解决一大堆问题的喜悦，但也有一些令人难忘的非技术时刻。

当詹姆斯告诉格雷格，他不应该越过他的头去寻找争论的意义时，他有一种具有深远影响的“屁股下面”的见解:“这很简单，格雷格。它不是在你头上，而是在你屁股底下。”同样是在马萨诸塞州的伯灵顿，一个专家小组在那里参加了凌晨3点新上映的《星球大战幽灵的威胁》的第二场放映。能在更适合睡觉的时间醒来的唯一理智的理由是，詹姆斯和妻子回加州看电影去了，他的妻子提前几个星期就买好了票。利用神奇的时区和早起的时间，我们看到了新发行的游戏。

得梅因的肉桂卷，大卫后来声称比他的头还大。这是一种夸张。每一卷都略小于大卫头的一半大小。

在渥太华的“死猫”会议上，格雷格声称当他带着耳朵痛去诊所时，医生可能会移除一只死猫。

“不礼貌用语”会议，也在渥太华。这个组织成为了计算机行业的八卦专栏，而我们对于被当作电影明星对待的兴奋之情，在这本书中根本无法表达。然而，令我们印象深刻的是，一个年纪足够大，认为格雷格是IBM的儿子的作家仍然经常写作。

1999年9月，该规范草案发布，供Java社区进程的参与者进行正式审查，并供从该团体网站(http://www.rtj.org)下载该规范的任何人进行非正式阅读。一九九九年十二月，经修订和扩展的文件在网页上公布，供公众审阅。公众谘询一直持续到2000年2月14日(是的，情人节)。然后对规范进行了最后一次修订，以回应公众的意见。

这项工作的第一个结果是你正在阅读的文件。IBM还为该规范生成了参考实现和测试套件。

致谢

读者应该认为这是一项真正的合作工作。许多人以不同的方式做出了贡献。与大多数传统出版的书籍不同，这项工作是工程师、高管、行政人员、营销和产品经理、行业顾问和来自全球二十多家公司和组织的大学教师努力和贡献的结果。它也是一种新的、独特的软件开发方法——Java社区过程的结果。

我们从头开始。许多技术贡献者聚集在国家标准与技术研究所的Lisa Carnahan构想并主持的一系列论坛上。其中一位作者格雷格·波利拉(Greg Bollella)与丽莎一起，在“未来作者”组织的早期建立中发挥了重要作用。他感谢他在IBM的经理Ruth Taylor、Rod Smith和Pat Sueltz，感谢他们(用他们的话说)成为低维护管理人员，并允许格雷格自由地追求他的目标。

Java社区过程是由Jim Mitchell、Ken Urquhart和其他人在Sun Microsystems开发的，目的是允许和促进计算机行业广泛参与Java™平台的开发。我们感谢他们以及Sun和其他公司的所有人，感谢他们对这个过程的最初建议进行了审查。Sun公司的嵌入式Java产品经理Vicki Shipkowitz还帮助了实时Java专家组，为RTSJ的演示和演示提供后勤支持。

Real-Time for Java专家组包括一个工程团队和一个顾问团队。这项工作的作者是主要工程师，我们真诚地感谢顾问们(前面提到过他们的名字)，感谢他们在早期设计阶段所做的努力以及对各种草案的审阅。在这一过程中，Ray Kamin, Wolfgang Pieb和Edward Wentworth替换了三位最初的顾问，我们也感谢他们的努力。

我们感谢所有这些人，特别是美国宇航局喷气推进实验室的Kirk Reinholtz，他在参与者和公众评审期间提交了评论。

我们感谢Sun Microsystems的Java系列编辑Lisa Friendly，以及Addison-Wesley的Mike Hendrickson、Sarah Weaver和Julie DiNicola，感谢他们为本书的编写所做的努力。

我们都感谢DISA的拉斯·理查兹对我们努力的支持。

我们感谢太阳微系统公司的Kevin Russell和Bill Foote，他们在NIST发起的需求阶段努力工作。

尽管他们还有很多工作要做，并且在他们实现RTSJ时可能会给我们带来更多的工作，我们还是要感谢IBM的参考实现团队。彼得·哈格尔领导着大卫·温特和吉姆·米克尔森的团队。Greg也感谢他们在他关于RTSJ的演讲中所使用的各种机器人演示上所做的努力。

格雷格想要亲自感谢他的论文指导老师凯文·杰夫对他的指导。

我们感谢Sun Microsystems的行政助理Robin Coron和IBM的Liu Feng，感谢他们的后勤支持。

格式说明

我们在Java源文件上使用javadoc来生成本书的大部分内容，因此许多对类、接口和方法名的引用都使用@link构造在(更典型的)html格式的输出中生成超链接。当然，单击html格式版本中的超链接将显示类的定义。当这本书被格式化为PDF格式时，我们试图保留这个超链接的特征，方法是在每出现一个名字时将其定义的页码作为末尾的下标。

# 第二版序言

规范的演变

RTSJ最早是在2000年年中发布的RTSJ 0.9版。0.9版本是专家组在1999年初到2000年Java One之前不久的工作成果。本文档并不是一个可用的规范，而是一个“供讨论的草案”。

在0.9版发布之后，规范方面的工作仍在继续，特别是参考实现的创建和参考实现的首次使用。这项工作产生了RTSJ的第一个正式版本，它是2002年1月从Java社区进程(JCP)中产生的。

当规范正式发布时，RTSJ的工作几乎没有减缓。专家组的一些成员加入了RTSJ社区的一些感兴趣的成员，作为技术解释委员会(TIC)，并继续改进规范。

RTSJ抽搐

Ben brosgol ada Core

大卫·霍尔姆斯技术咨询

鲁迪·贝利亚迪·施耐德自动化公司

道格洛克timesys

彼得种植timesys

Andy wellings约克大学

TIC工作的主要产品是RTSJ version 1.0.1，一个更完整的规范版本。在RTSJ 1.0版和1.0.1版之间，打印文档的大小从大约250页增加到大约500页，这表明了额外细节的数量。修订后的规范从2005年3月的JCP维护评审过程中产生，TIC继续进行工作。

本文档是RTSJ版本1.0.2。它继续了对1.0.1的解释工作，其中包含了1.0.1之间的许多变化。和1.0.2由RTSJ实现者和用户的澄清请求驱动。

RTSJ发展的下一个阶段是一个新的JSR。RTSJ 1.1版的专家组已经成立，关于RTSJ 1.1版的建议可以在Java Community Process站点的JSR 282页中找到

# 介绍

实时Java专家小组(RTJEG),召集在Java社区过程和jsr - 000001下,给出了产生规范的责任扩展Java语言规范的Java虚拟机规范,并提供一个应用程序编程接口,使创建、验证、分析、执行和管理Java线程的正确性条件包括时间性限制(也称为实时线程)。本文描述的指导原则RTJEG创建和使用在他们的工作,实时Java的描述要求主持下发达国家标准与技术研究所(NIST),和一个简短的,每个七个领域的高级描述完成专家组确定为需要改进的目标。

## 指导原则

指导原则是高层次的声明，它界定了RTJEG的工作范围，并为Java实时规范引入了兼容性要求。

对特定Java环境的适用性:RTSJ不应包括限制其在特定Java环境中的使用的规范，例如特定版本的Java开发工具包、嵌入式Java应用程序环境或Java 2 Micro Edition™。

向后兼容性:RTSJ不应阻止现有的、正确编写的、非实时Java程序在RTSJ的实现上执行。

写一次，在任何地方运行:RTSJ应该认识到“写一次，在任何地方运行”的重要性，但它也应该认识到为实时程序实现WORA的困难，不要试图以牺牲可预测性为代价来增加或维护二进制可移植性。

当前实践vs.高级特性:RTSJ应该解决当前的实时系统实践，并允许未来的实现包含高级特性。

可预测执行:RTSJ应在所有权衡中将可预测执行作为第一优先;这有时会以牺牲典型的通用计算性能度量为代价。

没有语法扩展:为了方便工具开发人员的工作，从而增加及时实现的可能性，RTSJ不应该引入新的关键字或对Java语言进行其他语法扩展。

允许实现决策的变化:RTJEG承认RTSJ的实现可能实现决策的不同,如使用高效或低效算法,时间和空间效率之间的权衡,包括调度算法不需要的最小实现,代码路径长度的变化和字节码的执行。RTSJ不应该强制要求算法或特定的时间常量，而是要求实现的语义必须满足。RTSJ为实现者提供了创建适合其客户需求的实现的灵活性。

## 7个增强区域的概述

在下面的七个部分中，我们将简要说明每个领域的方向。1999年3月下旬，8名主要工程师在加州门多西诺(Mendocino)举行的第一次会议上确定了这些方向，并在1999年9月下旬进行了进一步的澄清。

线程调度和调度:调度和显著的多样性的调度模型和识别不同的实时系统中的每个模型具有广泛的适用性,我们得出的结论是,方向调度规范是允许一个底层调度机制使用实时Java线程,但我们不会事先指定的本质(甚至一些)所有可能的调度机制。该规范的构造是为了允许实现提供非预期的调度算法。实现将允许以编程方式分配适合底层调度机制的参数，并为实时Java线程的创建、管理、准入和终止提供任何必要的方法。目前，我们还希望将特定的线程调度和调度机制绑定到一个实现上。但是，我们在线程调度框架中提供了足够的灵活性，允许该规范的未来版本在此版本的基础上构建，并允许动态加载调度策略模块。

为了适应当前的实践，RTSJ要求在所有实现中都有一个基本调度程序。所需的基本调度器将熟悉的实时系统程序员。它是基于优先级的，先发制人的，并且必须有至少28个独特的优先级。

内存管理:我们认识到自动内存管理是Java编程环境中一个特别重要的特性，我们寻求一个方向，使内存管理工作尽可能由底层系统自动实现，而不干扰编程任务。此外，我们知道存在许多自动内存管理算法，也称为垃圾收集(GC)，其中许多适用于实时编程风格和系统的某些类。为了适应不同的GC算法集，我们试图定义一个内存分配和回收规范:

独立于任何特定的GC算法，

允许程序精确地描述已实现的GC算法对实时Java线程的执行时间、抢占和调度的影响

允许对象的分配和回收不受任何GC算法的干扰。

同步和资源共享:逻辑通常需要串行访问资源。实时系统引入了额外的复杂性:控制优先级反转。我们已经决定，允许实时安全同步的干扰最小的规范是要求Java关键字synchronized的实现包含一个或多个算法，这些算法可以防止共享序列化资源的实时Java线程之间的优先级反转。我们还注意到，在某些情况下，使用synchronized关键字实现所需的优先级反转算法，既不能防止优先级反转，也不能允许线程具有逻辑上高于垃圾收集器的执行资格。我们提供了一组在这种情况下使用的无等待队列类。

异步事件处理:实时系统通常与现实世界密切交互。至于逻辑的执行，现实世界是异步的。因此，我们觉得有必要为编程规程加入有效的机制，以适应这种固有的异步性。RTSJ概括了Java语言的异步事件处理机制。所需的类表示可能发生的事情，以及当这些事情发生时执行的逻辑。一个值得注意的特性是，逻辑的执行是由已实现的调度器调度和分派的。

异步控制传输:有时现实世界的变化非常剧烈(而且是异步的)，因此当前的逻辑执行点应该立即有效地传输到另一个位置。RTSJ包含了一种机制，它扩展了Java的异常处理，允许应用程序以编程方式改变另一个Java线程的控制轨迹。值得注意的是，RTSJ限制了这种异步控制转移到特定编写的逻辑，假定其控制轨迹可能异步改变。

异步线程终止:同样，由于现实世界中有时会发生剧烈的异步更改，应用程序逻辑可能需要安排一个实时Java线程，以便快速和安全地将其控制转移到其最外层范围，从而以正常的方式结束。注意，与用于停止线程的传统的、不安全的和弃用的Java机制不同，RTSJ用于异步事件处理和控制传输的机制是安全的。

物理内存访问:虽然不是直接的实时问题，但是对于许多能够有效利用RTSJ实现的应用程序来说，物理内存访问是可取的。因此，我们定义了一个允许程序员字节级访问物理内存的类，以及一个允许在物理内存中构造对象的类。

# 设计

RTSJ包含了扩展语义的七个领域。本章介绍扩展。在每一相关章节的开头部分给出了进一步的细节、确切的要求和理由。这七个领域是按照它们与实时编程的相关性的近似顺序进行讨论的。然而，每个区域的语义和机制——调度、内存管理、同步、异步事件处理、异步控制传输、异步线程终止和物理内存访问——都是接受RTSJ作为可行的实时开发平台的关键。

## 调度

实时编程的关注点之一是确保机器指令序列的及时或可预测的执行。不同的调度方案对这些指令序列有不同的名称。通常使用的名称包括线程、任务、模块和块。RTSJ引入了可调度对象的概念。这些是基本调度程序管理的对象，RealtimeThread及其子类，以及AsyncEventHandler及其子类。

可调度对象的及时执行意味着程序员可以通过对程序的分析、在特定实现上或两者都测试程序来确定特定线程是否总能在给定的时效约束之前完成执行。这就是实时编程的本质:为计算的正确性条件添加时间约束。例如，对于一个计算两个数字的和的程序，只计算正确的算术答案可能不再是可以接受的，但答案必须在特定的时间之前计算。通常，时间限制是用相对时间或绝对时间表示的截止日期。

我们使用术语调度(或调度算法)来指为执行一组可调度对象(调度)而产生的序列(或排序)。这个计划试图优化一个特定的度量(度量系统满足时间限制的情况)。可行性分析确定一个计划是否具有可接受的度量值。例如，在硬实时系统中，典型的度量是“错过最后期限的数量”，而该度量的唯一可接受值是零。所谓的软实时系统使用其他度量(例如平均延迟)，并可能接受使用中的度量的各种值。

许多系统使用线程优先级来尝试确定调度。优先级通常是与可调度对象相关联的整数;这些整数向系统传递线程应该执行的顺序。优先级概念的概括是执行资格。我们使用分派一词来指系统中从准备运行的线程池中选择具有最高执行资格的线程的部分。在当前的实时系统实践中，优先级的分配通常是在程序员的控制下，而不是在系统控制下。RTSJ的基本调度器也让程序员控制优先级的分配。但是，基调度器还从其超类继承有助于确定可行性的方法。

对于基本调度器，可行性方法可以假定一个足够快的处理器来如期完成任何提议的负载。RTSJ期望基本调度器可以在特定的实现中子类化(例如，一个EDF调度器)，对于那些实现，可行性方法可以正确地指示在给定调度器下系统的实际可行性。注意，对于基本调度器，RTSJ与当前使用的大多数实时操作系统没有什么不同。

RTSJ需要许多类，其名称的格式为参数(例如SchedulingParameters)。其中一个参数类的实例拥有一个或多个可调度对象的特定资源需求特征。例如，SchedulingParameters的子类PriorityParameters包含基本调度器的执行资格度量，即优先级。在某个时候(在构造时或以后使用setter方法替换参数时)，参数类的实例被绑定到一个可调度对象。然后，可调度对象假定参数对象中值的特征。举个例子,如果一个PriorityParameters实例,在其优先级字段的值代表最高优先级可以是绑定到一个可调度的对象,该对象将假设时将执行它的特点是可以优先于所有其他可调度的对象(当然,除了那些具有最高优先级)。

RTSJ的编写是为了允许实现者在规范的实现中灵活地安装任意调度算法和可行性分析算法。我们这样做是因为RTJEG了解实时系统行业在调度方面有广泛的不同要求。使用Java平台可以帮助生成编写一次但能够在许多不同的计算平台上执行的代码(称为编写一次，在任何地方运行)。RTSJ既有助于实现这一目标，也不利于实现这一目标。RTSJ对所需优先级调度器的严格规范对于时间关键代码的可移植性至关重要，但是RTSJ允许并支持不可移植的特定平台调度器。

## 内存管理

由于垃圾收集器引入了不可预测的延迟，垃圾收集的内存堆一直被认为是实时编程的障碍。RTSJ通过对内存模型提供几个扩展来解决这个问题，这些扩展以一种不干扰实时代码提供确定性行为的能力的方式支持内存管理。这个目标是通过允许在垃圾收集堆之外为短期和长期生存的对象分配对象来实现的。

内存区域

RTSJ引入了内存区域的概念。内存区域表示可用于分配对象的内存区域。有些内存区域存在于堆之外，并对系统和垃圾收集器可能对在堆内分配的对象执行的操作进行了限制。某些内存区域中的对象永远不会被垃圾收集;但是，垃圾收集器必须能够扫描这些内存区域，以查找对堆内任何对象的引用，以保持堆的完整性。

有四种基本类型的记忆区域:

作用域内存提供了一种比堆栈分配对象更通用的机制，用于管理由作用域定义生命周期的对象。

物理内存允许在具有特殊重要特征的特定物理内存区域中创建对象，例如具有更快访问速度的内存。

不朽内存表示一个包含对象的内存区域，这些对象可以被任何可调度对象引用而没有异常或垃圾收集延迟，特别是包括无堆实时线程和无堆异步事件处理程序。

堆内存表示作为堆的内存区域。RTSJ不会更改堆上对象的生存期的决定因素。寿命仍然是由能见度决定的。

作用域内存

RTSJ引入了作用域内存的概念。内存作用域用于为在其内分配的任何对象的生存期划定界限。当输入一个作用域时，每次使用new都会导致从活动内存作用域分配内存。范围可以显式输入，也可以附加到可调度对象，该对象将在执行对象的run()方法之前有效地进入范围。

当作用域内的对象不能被引用时，将丢弃作用域内的内容。这是通过类似于引用计数作用域的技术来实现的。一个一致的实现可能会维护对每个内存区域的外部引用数量的计数。通过使用MemoryArea的enter()方法输入一个新的作用域，通过使用特定作用域内存区域创建一个可调度对象，或者通过打开一个内部作用域，可以增加作用域内存区域的引用计数。当从enter()方法返回时，当使用ScopedMemory的可调度对象终止时，或者当内部作用域从其enter()方法返回时，作用域内存区域的引用计数将减少。当计数降为0时，内存中每个对象的finalize方法将被执行到完成。作用域的重用被阻塞，直到结束。

作用域可以嵌套。当进入嵌套作用域时，所有后续分配都从与新作用域关联的内存中获取。当退出嵌套作用域时，将恢复前一个作用域，并再次从该作用域获取后续的分配。

由于作用域对象的生存期，有必要通过一组受限制的赋值规则来限制对作用域对象的引用。不能将对作用域对象的引用分配给来自外部作用域的变量，也不能分配给堆或不朽区域中的对象字段。对作用域对象的引用只能分配到相同的作用域或内部作用域。虚拟机必须检测到非法的赋值尝试，并在它们发生时抛出适当的异常。

在选择作用域内存类型方面提供的灵活性允许应用程序使用具有适合于特定语法定义的代码区域的特征的内存区域。

不朽内存

不死内存是应用程序中所有可调度对象和线程之间共享的内存资源。在ImmortalMemory中分配的对象总是对非堆线程和异步事件处理程序可用，而不可能延迟垃圾收集。

预算分配

对于使用内存区域为可调度对象提供内存分配预算，RTSJ还提供了有限的支持。可以在创建单个可调度对象时指定它们的最大内存区域消耗和最大分配率。

## 同步

条款

为本节的目的，对优先一词的解释应比常规用法稍微宽松一些。特别地，术语最高优先级线程仅仅表示最合适的线程——调度程序将在所有准备运行的线程中选择的线程——并没有必要假定严格的基于优先级的调度机制。

等待队列

等待获取资源的线程和异步事件处理程序必须按执行资格顺序释放。这既适用于处理器，也适用于同步块。如果在活动调度策略下可能存在具有相同执行资格的可调度对象，那么这些可调度对象将按FIFO顺序被唤醒。例如:

等待进入同步块的线程按照执行资格顺序被授予对同步块的访问权。

准备运行的阻塞线程按执行资格顺序获得对处理器的访问权。

一个线程的执行资格由它自己显式地设置，或者另一个线程按照执行资格顺序被授予对处理器的访问权。

执行yield的线程将在等待具有相同执行资格的线程之后获得对处理器的访问权。

为支持具有更高执行资格的线程而抢占的线程可以在特定实现决定的任何时间被授予对处理器的访问权。实现需要提供文档，准确说明用于授予此类访问权限的算法。

避免优先级反转

任何符合标准的实现都必须提供一个具有默认行为的同步原语实现，以确保没有无限制的优先级反转。此外，如果代码在实现中运行，这必须适用于代码以及实时线程。默认情况下必须实现优先级继承协议。优先级继承协议是实时调度文献中比较著名的算法，其效果如下。如果线程t1试图获得一个低优先级线程持有的锁t2,然后t2的首要任务是提高t1只要t2持有锁(递归地如果t2本身就是等待获得一个更低优先级线程持有的锁)。

该规范还提供了一种机制，程序员可以通过该机制重写默认的系统范围策略，或者控制用于特定监视器的策略，前提是该策略得到了实现的支持。监视控制策略规范是可扩展的，因此未来的实现可以添加新的机制。

第二种策略，优先级上限仿真协议(或最高锁柜协议)，也为支持它的系统指定。该协议也是文献中比较知名的算法;稍微简化一下，其效果如下:

监视器在创建时被赋予一个“优先级上限”;程序员应该选择任何可能试图进入监视器的线程的最高优先级。

一旦线程进入同步代码，它的(活动的)优先级就会被提升到监视器的最高优先级。如果由于编程错误，线程的基本优先级高于它试图进入的监视器的上限，则抛出异常。

离开监视器时，线程的活动优先级被重置。在简单的情况下，它将被设置为线程之前的活动优先级，但在某些情况下(例如线程在监视器中动态改变了其基本优先级)，一个不同的值是可能的

注意,尽管RTSJ需要短命的可调度的执行对象不能被推迟被垃圾收集代表优先级可调度的对象,应用程序可以导致no-heap可调度的对象等垃圾收集由一个使用线程之间的同步使用对象或可调度的对象和一个短命的可调度的对象。RTSJ提供了无等待队列类，为普通Java线程和无堆实时线程访问的对象提供受保护的、非阻塞的共享访问。显式提供这些类是为了支持非堆可调度对象的实时执行与常规Java线程或使用堆的可调度对象之间的通信。

决定论

符合标准的实现应提供一个固定的上限时间，以进入一个未锁定的监视器的同步块。

## 异步事件处理

异步事件工具包括两个类:AsyncEvent和AsyncEventHandler。AsyncEvent对象表示可能发生的事情，如POSIX信号、硬件中断或计算机事件(如飞机进入指定区域)。当其中一个事件发生时(由调用的fire()方法指示)，将调度AsyncEventHandler的关联实例，并调用handleAsyncEvent()方法，从而执行所需的逻辑。此外，提供了AsyncEvent上的方法来管理与AsyncEvent实例相关联的AsyncEventHandler实例集。

AsyncEventHandler的实例可以看作类似于线程。它是一个可运行对象:当事件触发时，将调度相关的处理程序，并调用handleAsyncEvent()方法。AsyncEventHandler与简单的Runnable的区别在于，AsyncEventHandler有释放参数、调度参数和内存参数的关联实例，它们在关联的AsyncEvent被触发后控制处理程序的实际执行。当事件被触发时，处理程序被异步执行，根据相关的ReleaseParameters和SchedulingParameters对象进行调度，就像处理程序刚刚被分配给它自己的线程一样。它的目的是使系统能够很好地处理存在大量AsyncEvent和AsyncEventHandler实例(数以万计)的情况。被触发的(进程中)处理程序的数量预计会更少。

AsyncEvent的一种特殊形式是Timer类，它表示由时间驱动的事件。计时器有两种形式:OneShotTimer和PeriodicTimer。OneShotTimer的实例在指定的时间触发一次。周期性定时器首先在指定的时间触发，然后按照指定的时间间隔周期性地触发。

计时器由时钟对象驱动。有一个特殊的时钟对象，Clock. getrealtimeclock()，它表示实时时钟。Clock类可以扩展为表示底层系统可能提供的其他时钟(比如某个粒度的执行时间时钟)。

## 异步控制转移

实时程序员经常面临这样一种情况:算法的计算成本是高度可变的，算法是迭代的，并且算法在每次迭代中产生连续的精炼结果。如果系统,在开始计算之前,只能确定一次绑定执行多长时间计算(例如,每个迭代是高度可变的成本和所需的最小延迟终止计算和接收最后一致的结果远低于大约一半的意思是迭代的成本),然后，在已知时间期限到期时，异步地将控制从计算转移到结果传输代码是一种方便的编程风格。RTSJ支持这种和其他风格的编程，这种传输是方便的，具有一个称为异步控制传输(ATC)的特性。

RTSJ的空中交通管制的方法是基于几个指导原则，非正式地列出了以下清单。

方法论的原则

一种方法必须明确表明其对ATC的敏感性。由于遗留代码或库方法可能是在没有ATC的情况下编写的，因此在默认情况下，ATC必须关闭(更准确地说，只要控制在这些代码中，就必须延迟)。

即使一个方法允许空中交通管制，一些代码段必须执行到完成，因此空中交通管制延迟在这些部分。这些atc延迟的部分是同步方法、静态初始化器和同步语句。

响应ATC的代码不会返回到ATC触发的可调度对象中的点;也就是说，空中交通管制是一种无条件的控制权转移。恢复语义(它将控制从处理程序返回到中断点)是不需要的，因为它们可以通过其他机制(特别是AsyncEventHandler)实现。

替代原则

需要一种机制，通过这种机制，ATC可以显式地在目标可调度对象中触发。这个触发可以是直接的(来自源线程或可调度对象)，也可以是间接的(通过异步事件处理程序)。

必须能够基于任何异步事件触发ATC，包括外部发生的事件或从另一个线程或可调度对象触发的显式事件。特别是，它必须是可能的基础上一个计时器的ATC离开。

通过ATC，必须能够中止实时线程，但要以不带有线程类的stop()和destroy()方法的危险的方式。

语义原则

如果ATC是通过异常处理建模的，那么必须有某种方法来确保异步异常只被预期的处理程序捕获，而不是被(例如)碰巧在传播路径上的通用处理程序捕获。

嵌套的ATCs必须正常工作。例如，考虑两个嵌套的基于atc的计时器，并假设外部计时器的超时时间比嵌套的内部计时器短。如果外部计时器超时，而控制在内部计时器的嵌套代码中，那么嵌套代码必须被中止(只要它在ATC-deferred部分之外)，然后控制必须转移到外部计时器的适当catch子句。在嵌套代码中处理外部超时或等待较长的(嵌套的)计时器的实现是不正确的。

务实的原则

对于常见的情况，比如定时器处理程序和实时线程终止，应该有一些简单的习惯用法。

如果带有超时的代码在计时器到期之前完成，则需要自动停止计时器并将其资源返回给系统。

## 异步实时线程终止

尽管不只是实时问题，许多事件驱动的计算机系统与外部现实世界的非计算机系统(例如，人、机器、控制进程等)紧密交互，可能需要在其计算行为的模式变化，作为非计算机现实世界系统的重大变化的结果。当外部实时系统发生变化，线程不再有用时，编写异常终止的线程将是很方便的。如果没有这个功能，就必须以这样的方式对线程或线程集进行编码，以便它们的计算行为能够预测外部系统可能状态之间的所有可能转换。编写线程来为外部系统的一个(或极少的)可能状态进行计算性协作是一项更容易的设计任务。当外部系统状态转换,然后计算行为的变化可能是由oracle,终止一组线程用于旧的外部系统,并调用一组新的线程适合外部系统的新状态。由于外部系统可能的状态转换仅在oracle中编码，而不是在每个线程中，因此整体系统设计更容易。

Java语言的早期版本提供了实现这些效果的机制:特别是类Thread中的stop()和destroy()方法。然而，由于stop()可能使共享对象处于不一致的状态，所以stop()已被弃用。使用destroy()可能会导致死锁(如果线程在持有锁时被销毁)，尽管直到Java规范1.5版才弃用它，但它的使用一直不被鼓励。RTSJ的目标是满足异步线程终止的要求，同时不引入stop()或destroy()方法的危险。

RTSJ通过结合异步事件处理和异步传输控制机制，实现了安全的异步实时线程终止。要创建这样一组实时线程，请考虑以下步骤:

使实时线程的所有应用程序方法都可中断

创建一个oracle，通过将大量异步事件处理程序绑定到适当模式更改时发生的事件，从而监视外部世界

处理程序是否在每个受更改影响的实时线程上调用interrupt()

在处理程序调用interrupt()之后，让它们创建一组新的实时线程，以适应外部世界的当前状态。

发生这种情况的结果是，通过将控制权转移给适当的catch子句，导致每个可中断方法异常中止。最终，实时线程的run()方法将正常完成。

这种习惯用法提供了实时线程的快速(如果编码是这样的话)但有序的清理和终止。请注意，oracle可以包含任意多或少的异步事件处理程序。

## 物理内存访问

RTSJ为希望直接从Java语言编写的代码中访问物理内存的程序员定义了类。RawMemoryAccess定义了一些方法，允许程序员构造一个表示物理地址范围的对象。然后通过对象的get[type]()和set[type]()方法来访问物理内存，其中类型表示单词大小，即byte、short、int、long、float和double。除了set[type]()和get[type]()方法之外，没有隐含其他语义。VTPhysicalMemory、LTPhysicalMemory和ImmortalPhysicalMemory类允许程序员构造一个表示物理内存地址范围的对象。当该对象用作MemoryArea时，可以在物理内存中使用new关键字构造其他对象。

PhysicalMemoryManager可供使用的各种物理内存访问对象(VTPhysicalMemory, LTPhysicalMemory, ImmortalPhysicalMemory、RawMemoryAccess RawMemoryFloatAccess)创建对象的正确类型绑定到适当的特色区域的物理内存,或者用适当的访问器的行为。可能被指定的特征的例子有:DMA内存，具有字节交换的访问器，等等。oem可以提供PhysicalMemoryTypeFilter类，允许指定内存设备的附加特征。

### 原始内存访问

RawMemoryAccess的一个实例将一系列物理内存建模为一个固定的字节序列。完整的访问器方法允许通过与基值的偏移量访问物理区域的内容，这些偏移量被解释为字节、短、int或长数据值或这些类型的数组。

偏移量是指定多字节值的最重要字节还是最不重要字节，会受到类RealtimeSystem中的BYTE\_ORDER静态变量的影响，可能会通过与底层物理内存类型相关联的字节交换属性进行修改。

RawMemoryAccess类允许实时程序实现设备驱动程序、内存映射I/O、闪存、电池支持的RAM和类似的低级软件。

原始内存区域不能包含对Java对象的引用。这样的功能是不安全的(因为它可以用来阻止Java的类型检查)，而且容易出错(因为它对Java编译器做出的特定表示选择很敏感)。

### 物理内存区域

在许多情况下，由于性能或其他原因，需要可预测执行RTSJ的系统还需要访问特定地址的各种内存。考虑一个系统，在该系统中可以通过编程获得非常快的静态RAM。可以优化性能的设计可能希望在快速静态RAM中放置各种经常使用的Java对象。VTPhysicalMemory、LTPhysicalMemory和ImmortalPhysicalMemory类允许程序员具有这种灵活性。程序员将在快速RAM占用的内存地址上构造一个物理内存对象

## Exceptions

RTSJ引入了几个新的异常，以及围绕控制和内存分配器的异步传输异常的一些新的处理方法。

# 最低要求

## 要求和惯例

本规范的基本要求是：

除非本规范特别要求，否则任何实现都应完全符合Java平台配置。

除非本章另有说明，否则本规范的任何实现都应实现本规范中的所有类和方法。特别是，每个实现都必须包含PriorityScheduler类的一致性实现。

这个javax.realtime文件包装不得包含本规范中未包含的公共或受保护的方法。

JVM的实现方式不应允许在其实现的任何调度交互中实现无限优先级反转。

根据通常的假设javax.realtime文件可以安全地由多个线程同时使用，除非另有说明。

不需要PhysicalMemoryTypeFilter的特定实例，但每个实现必须至少支持一个这样的实例，其特征是它支持对实现可以访问的物理内存范围的访问。

在非周期参数、PhysicalMemoryManager、SporadicParameters、RealtimeSystem和PriorityScheduler中找到的静态最终值必须由实现处理，以便在编译时无法解析它们的值。

本规范的许多方面设置了最低要求，但允许在其实现中使用实现自由度。例如，所需的优先级调度器需要至少28个连续编号的实时优先级。但是，它没有指定最大优先级和最小优先级的数值。鼓励实现提供尽可能多的实时优先级。

除非另有规定，否则当本规范要求创建对象时，将在当前分配上下文中创建对象。

## 可选设施

基于本规范的扩展没有限制，除了在系统中只能实现符合本规范未来版本的扩展之外javax.realtime文件包裹。

本规范中包括几个可选扩展。应用程序不能在每个实现中都依赖于这些设施，但是如果实现了可选设施，则应用程序可以依赖它来执行此处指定的操作。这些扩展是：

|  |  |
| --- | --- |
| 成本强制执行 | 允许应用程序控制可调度对象的处理器利用率。 |
| 处理组强制 | 允许应用程序控制一组可调度对象的处理器利用率 |
| 处理组的截止日期小于期间 | 允许应用程序指定小于处理组周期的处理组截止时间 |
| 优先级上限仿真协议 | 一种避免优先级反转的优先级继承方法 |
| 对原始内存的原子访问 | 大多数原子访问是可选的。该实现可以在形式的系统属性中提供原始存储器访问特性javax.realtime.atomicaccess\_<xxx>。 |
| 堆分配上的分配率强制 | 允许应用程序限制可调度对象在堆中创建对象的速率。 |

ProcessingGroupParameters类仅在支持处理组强制选项的系统上起作用。成本强制和成本超支处理程序仅在支持成本强制选项的系统上起作用。如果支持处理组强制，则ProcessingGroupParameters必须按指定的方式运行。如果支持成本强制，则成本强制和成本超支处理程序必须按指定的方式运行。

如果不支持处理组截止时间小于period，则传递给ProcessingGroupParameters的构造函数及其setDeadline方法的值将被约束为等于period。如果支持该选项，则必须支持处理组截止日期小于期间，并按指定方式运行。

如果支持优先级上限模拟，则必须按指定实现优先级上限模拟。如果不支持优先级上限模拟，则必须存在PriorityCeilingEmulation，但实现可能不允许将其用作监视控制策略。

如果支持堆分配率强制，则必须按指定的方式实现。如果不支持堆分配率强制，则必须检查MemoryParameters的分配率属性的有效性，否则将被实现忽略。

以下语义对于专门作为开发工具设计和许可的RTSJ实现是可选的：

* + 优先级调度器不需要支持固定优先级抢占式调度或优先级继承。这并不能成为实现完全支持相关api的借口。它只将底层调度器所需的行为降低到Java规范中调度器的级别，扩展到至少28个优先级。
  + 不需要支持超出Java规范要求的语义约束时间。具体来说，垃圾收集可能会延迟任何线程而不绑定，并且允许延迟传递异步中断的异常，包括从不传递异常。但是，请注意，如果交付了除泛型AIE之外的任何AIE，那么它必须满足AIE语义，并且除抢占之外的所有堆内存相关语义仍然完全有效。此外，放松的计时并不意味着放松的排序。例如，必须完全实现作用域内存的语义。
  + 改变标准Java方法行为的RTSJ语义，例如Thread.setPriority设置优先级以及线程中断-开发工具不需要，但必须记录与RTSJ的偏差，而且实现必须能够在每次这些方法中的一个偏离标准RTSJ行为时生成运行时警告。
  + 这些宽松的需求为RTSJ开发系统工具的实现奠定了基础。开发工具可以选择实现不需要的语义。

## 有条件要求的设施

如果底层硬件和软件允许，实现必须支持有条件要求的设施。本规范包括三个有条件要求的设施：

|  |  |
| --- | --- |
| POSIXSignalHandler | 该类应在支持POSIX信号的每个平台上实现 |
| RawMemoryFloatAccess | 这应在每个平台上实现，其中基本JVM包括对浮点和双类型的支持。 |
| Mapping memory | 如果系统支持地址转换，则实现应支持原始内存访问类的内存映射特性。 |

如果不支持POSIX信号，则必须不存在POSIXSignalHandler类。如果支持POSIX信号，则必须按照指定实现POSIXSignalHandler。

如果平台不支持浮点，则RawMemoryFloatAccess不能存在。如果支持浮点，则必须按指定实现RawMemoryFloatAccess。

## 所需文件

RTSJ的每个实现都需要为以下几种行为提供文档：

1. 如果可行性测试算法不是默认的，记录可行性测试算法。
2. 如果应用程序可以使用除基本优先级调度器以外的调度器，请记录调度器的行为及其与其他调度器的交互，如“调度”一章中所述。记录构成调度程序的可调度对象的类的列表，除非该列表与基本调度程序的可调度对象列表相同。如果在非堆上下文中对调度程序的使用有限制，请记录这些限制。
3. 被高优先级可调度对象抢占的可调度对象被放置在队列中，以获得其活动优先级，位置由实现确定。如果抢占的可调度对象未放置在相应队列的前面，则实现必须记录用于此类放置的算法。在本规范的未来版本中，可能需要在队列前面放置。
4. 如果实现支持成本强制，那么实现需要记录更新当前CPU消耗的粒度。
5. 任何物理内存类型筛选器实现的内存映射都必须记录在案，除非它是连续字节的简单顺序映射。
6. 实现必须完整地记录GarbageCollector的任何子类的行为。
7. 提供本规范中未详述的任何MonitorControl子类的实现必须记录它们的效果，特别是对于优先级反转控制和调度程序（如果有的话）不支持新策略。
8. 如果由于优先级反转避免算法而失去“增强”优先级时，可调度对象没有放在其新队列的前面，则实现必须记录队列行为。
9. 对于基本调度器以外的任何可用调度器，实现必须记录同步的语义与为默认优先级继承监视器控制策略定义的规则之间的差异（如果有的话）。它必须提供与同步一章中基本优先级调度器的语义等效的优先级继承（如果支持，还包括优先级上限仿真协议）的新调度器行为的文档。如果在无堆上下文中对调度程序的使用有限制，则文档必须详细说明这些限制对每个rtsjapi的影响。
10. 对于某些参考体系结构，必须记录由于释放关联的AsyncEventHandler（假设没有更高优先级的可调度对象可运行）时发生绑定而触发AsyncEvent之间的最坏情况响应间隔。
11. 对于某些参考体系结构，必须记录在ATC启用线程上触发异步中断exception和首次传递该异常（假设没有更高优先级可调度对象可运行）之间的间隔。
12. 如果支持成本强制，并且实现将为作用域内存中的对象运行终结器的成本分配给任何可调度对象，而不是通过离开作用域而导致作用域引用计数降至零的对象，则应记录分配成本的规则。
13. 如果支持成本强制执行，且强制执行（因成本超支而受阻）可延迟超过强制执行时间粒度，则应记录此类延迟的最大值。
14. 如果RealtimeSecurity的实现比所需的实现更具限制性，或者具有运行时配置选项，则应记录这些特性。
15. 对于每个支持的时钟，文档必须指定分辨率是否可设置，如果可设置，文档必须指明支持的值。
16. 如果实现包括除所需实时时钟以外的任何时钟，则它们的文档必须指明这些时钟可以在什么上下文中使用。如果它们不能在无堆上下文中使用，文档必须详细说明将时钟或使用时钟的时间传递给无堆可调度对象的结果。

## 习惯

在整个RTSJ中，当我们使用code这个词时，我们指的是用Java编程语言编写的代码。当我们在RTSJ中提到Java语言时，它也指Java编程语言。在RTSJ中使用heap这个术语是指Java语言运行时使用的堆。

## 定义

A *thread* is an instance of the java.lang.Thread class.

A *real-time thread* is an instance of the javax.realtime.RealtimeThread class.

A *Java thread* is a thread that is not a real-time thread.

A *no-heap real-time thread* is an instance of the javax.realtime.NoHeapRealtimeThread class.

An *asynchronous event handler* is an instance of the javax.realtime.AsyncEventHandler class.

术语Schedulable object不同于术语Schedulable object（SO）。实现可调度接口的每个对象都可以称为可调度对象，但是只有被基本调度器识别为可调度实体的对象才是相对于该调度器的可调度对象。基本调度器的可调度对象集包含RealtimeThread和AsyncEventHandler的实例。其他调度器可能支持一组不同的可调度对象，但本规范仅定义基本调度器的行为，因此术语可调度对象应理解为“可由基本调度器调度”

# 标准Java类

在一些情况下，RTSJ的语义会影响标准Java类库中类的语义。明确地：

* 中优先级的set和get方法java.lang.Thread线程用于实时线程。
* ThreadGroup类相对于实时线程的行为。
* 线程中与线程组相关的方法应用于实时线程时的行为。

## 优先

方法中的setPriority和getPriority方法java.lang.Thread线程是最终决定。因此，实时线程类无法重写它们并修改它们的行为以满足RTSJ调度器的要求。带来java.lang.Thread线程类与实时子类一致，getPriority和setPriority方法的语义修改如下：

* Thread.setPriority():

1. 使用Thread.setPriority设置优先级（）不应影响由优先级公式和优先级继承控制的优先级反转避免算法的正确性。由于调用Thread.setPriority设置优先级（）由来自同步的语义控制。
2. 实时线程可以使用setPriority访问实时线程可用的扩展优先级范围。如果实时线程的优先级参数对象未共享，则setPriority的行为有效，好像它包含代码段：PriorityParameters pp=getSchedulingParameters（）；pp.setPriority（新优先级）；
3. 如果实时线程的priority parameters对象与其他可调度对象共享，setPriority必须为线程提供一个非共享PriorityParameters实例，该实例与实时线程对象分配在同一内存区域中，并包含新的priority值。
4. 如果线程是实时线程，并且新优先级超出实时线程调度程序允许的范围，则setPriority抛出IllegalArgumentException。
5. 如果线程是实时线程且其调度参数对象不是PriorityParameters的实例，setPriority将引发ClassCastException。

* Thread.getPriority():

1. 在实时线程上使用时，getPriority的行为就像它包含代码段一样有效：（（PriorityParameters）t.getSchedulingParameters（））.getPriority（）；
2. 如果调度参数不是PriorityParameters类型，则抛出一个ClassCastException。

所有受支持的监视控制策略都必须应用于Java线程以及所有可调度对象。

## 线程组

线程组以一个基本的ThreadGroup对象为根，这个对象可以在堆或不朽内存中创建。所有线程组对象都包含对其所有成员线程和子组的引用，以及对其父组的引用。由于堆和不朽内存不能保存对作用域内存的引用，因此永远不能在作用域内存中分配线程组。因此，在作用域内存中分配的任何线程都不能从任何线程组引用，因此这些线程不是任何线程组的一部分，并且将持有空的线程组引用。类似地，noheapreadimethread不能是堆分配的线程组的成员。

1. 枚举、中断、停止、恢复或挂起线程组时，不包括具有空线程组的实时线程。但是，当当前线程是具有空线程组的实时线程时：
   1. 这个Thread.枚举类方法返回整数1，并用当前实时线程填充其数组参数。
2. 从实时线程创建的Java线程继承实时线程的线程组（如果有）；否则会尝试将其添加到应用程序根线程组。如果不允许Java线程使用应用程序根线程组，则构造函数应抛出SecurityException。
3. 异步事件处理程序创建的Java线程的线程组被分配，就像它是由没有线程组的实时线程创建的一样（如2所述）
4. 无法在作用域内存中创建线程组。构造函数将抛出IllegalAssignmentError。
5. 线程组中设置的优先级限制对实时线程没有影响。
6. 除了前面指定的以外，实时线程与父线程类具有相同的线程组成员身份规则。

# 变化

# 线程

# 定时器

# 内存管理

# 同步

# 时间

# 时钟和计时器

# 异步

# 系统和选项

# 异常

# 已弃用

# 全规格许可证

# 评估规范许可证