文章编号:1008-0570(2010)11-3-0176-03

# 基于公平策略的 Linux 2.6 调度算法与应用分析

*Linux 2.6 Fair Strategy Based Scheduler and Its Application Analysis*

(苏州大学) 李建萍 陆建德
LI Jian-ping LU Jian-de

**摘要:** 传统 Linux 2.4 调度存在诸多缺陷,Linux 2.6 各版内核先后采用 O(1)及 CFS 调度,大大改进了原来调度中存在的问题,O(1) 与 CFS 调度各有其特点,本文对这些调度器进行深入的剖析与比较,并重点解析 CFS 调度,笔者对各调度策略的实现过程及适合的应用做了详细的探讨。

**关键词: O(1); CFS; 公平; 优先级调度; 红黑树**
**中图分类号: TP393** **文献标识码: A**

**Abstract:** Traditional Linux 2.4 schedulers existed some defects and several Linux 2.6 kernel distributions adopted either O (1) or CFS scheduler early and late,which obviously improving the original performance of CPU scheduling. Either O (1) or CFS scheduler has its own characteristics. This paper analyzed and compared these schedulers in depth,especially focused on parsing CFS scheduling. The implementation of various scheduling strategy and its appropriate application have also been thoroughly discussed by this author.

**Key words: O(1); CFS; fair; priority scheduling; red-black tree**

技术创新

## 1 引言

Linux 作为主流的操作系统,其内核调度算法历来是人们改进系统性能的研究热点。纵观 Linux 调度器的发展,大致经历了三个阶段:1992 年,Linus 发布 O (n) 调度算法, 其一直发展到 2.4.18 内核,基本未有大的变化;随后,在 2.6 内核中改以由 Ingo-Molar 设计并实现的 O(1)调度算法,它与以往内核版本相比,性能获得了很大改善;直到 2.6.23 内核及其后续版本,才产生了新的基于完全公平策略的调度器 CFS (Completely Fair Scheduler),此时 Linux 的内核调度算法具有了革命性的改变。

笔者对 Linux 2.4、2.6 各版内核中的调度器代码分别进行了解读与剖析,做了许多比较工作。对 Linux 2.6 内核调度的分析表明,Linux 2.6 内核采用的调度更复杂且有效率,2.6 各版内核先后采用 O(1)、RSDL 及 CFS 调度,这些新的调度策略大大改进了 2.4 内核调度性能,2.6 内核调度算法所花费的时间与当前系统中的进程个数无关, 在高负载的情况下执行得极其出色,并且很好解决了多处理器的扩展性;Linux 2.6 支持内核态抢占,对实时进程的支持更好, 对交互式进程和批处理式进程也能更好地区分处理。在本文中,笔者详细分析 Linux 2.6 内核中最有代表性的 O(1)和 CFS 调度算法,并对各种调度策略适用的应用环境进行了测试。分析与测试结果表明,2.6 内核采用的 O(1)调度由于采用优先级队列和查找位图, 其挑选进程和插入进程的速度更快,更加适合实时进程,而 2.6 内核采用的 CFS 由于红黑树的动态特性在处理批处理进程和交互式进程时则更有优势。

## 2 Linux 2.6 O(1)调度工作原理及特点分析

### 2.1 Linux 2.4 内核调度策略的回顾与缺陷

Linux 2.4 内核调度采用基于优先级的设计, 调度算法 Pick next 较简单:仅对全局就绪队列 runqueue 中所有进程的优先级依次进行比较,选择最高优先级进程作为下一个被调度进程。每个进程被创建时赋予一个时间片, 时钟中断递减当前运行进程的时间片, 时间片用完时进程必须等待重新赋予时间片才能有机会运行。Linux 2.4 中只有当所有 RUNNING 进程的时间片都用完之后,才对所有进程重新分配时间片。因此 2.4 内核存在某些缺陷:

* 系统调度算法复杂性 O(n), 开销呈线性增长;
* 仅一个全局就绪进程队列,多处理器扩展性差;
* 交互式进程优化并不完善: 由于交互式进程比批处理进程更频繁处于 SUSPENDED 状态。而有些批处理进程虽然没有用户交互,但是也会频繁进行 I/O 操作,虽不需快速用户响应,还是被提高了优先级,这会影响真正交互式进程的响应时间。
* 时间片重算制约了多处理器效率, 当大多数就绪进程的时间片皆用完且未重新分配时, 对称多处理器 SMP 系统的有些处理器会一直处于空闲状态。
* 对实时进程的支持不够:Linux2.4 内核是非抢占的, 当进程处于内核态时不会发生抢占, 这对于真正的实时应用不能接受。

### 2.2 Linux 2.6 O(1)调度策略的分析

对 Linux 2.6 内核调度的分析表明,O(1)调度算法基于每个 CPU 分配时间片,并且取消了全局同步和重算循环。每个处理器有两个数组: 活动就绪进程队列数组和不活跃就绪进程队列数组。每个数组中有 140 个就绪进程队列(runqueue),每个队列皆为先进先出 ( FIFO),并由一位图指示队列是否空。这样,挑选进程时只需通过 find_first_bit 找到第一个不为空的队列, 并取队首的进程即可。若一个进程消耗完其"时间片", 就进入不活跃就绪进程数组的相应队列的队尾。当所有的进程皆"耗尽"各自的"时间片",交换活跃与不活跃就绪进程队列数组的指针即可, 不需任何其他开销。以上的 O(1)算法不管队列中有多少个

李建萍: 硕士研究生

就绪进程，挑选就绪进程的速度是一定的。

O(1)调度算法具有常量性，在O(1)中，就绪队列被定义为一个很复杂的数据结构runqueue，每一个CPU都将维护一个自己的就绪队列，该结构定义了变量prio_array_t *active, *expired, arrays,(prio_array优先进程队列)。每个CPU的就绪队列按时间片是否用完分为两部分，分别通过active指针和expired指针访问，active指向时间片没用完、当前可被调度的就绪进程，expired指向时间片已用完的就绪进程。每一类就绪进程都用一个结构prio_array表示：

```
struct prio_array {
    unsigned int nr_active;     /* 当前活跃的进程总数 */
    unsigned long bitmap [BITMAP_SIZE]; /* 活跃进程的位图 */
    struct list_head queue[MAX_PRIO]; /* 各个优先级队列的头指针组成的数组 */
};
```

prio_array中包含一个就绪队列数组，数组的索引是进程的优先级(共140级)，相同优先级的进程放置在相应数组元素的链表队列中。调度时直接给出就绪队列active中具有最高优先级的链表中的第一项作为候选进程。为了加速寻找存在就绪进程的链表，O(1)核心又建立了一个位映射数组来对应每一个优先级链表，如果该优先级链表非空，则对应位为1，否则为0，以便快速定位第一个非空的就绪进程链表。arrays数组是两类就绪队列的容器，active和expired分别指向其中一个。active中的进程一旦用完了自己的时间片，就被转移到expired中，并设置好新的初始时间片；而当active为空时，则表示当前所有进程的时间片都消耗完了，此时，active和expired进行一次对调，重新开始下一轮的时间片递减过程。

采用这种将集中计算过程分散进行的算法，保证了O(1)调度器运行的时间上限，同时在内存中保留更加丰富的信息的做法也加速了候选进程的定位过程。

在O(1)中，实时进程和非实时进程的优先级计算方法更科学。Linux内核将进程分为两类：实时进程(SCHED_RR和SCHED_FIFO)和非实时进程(SCHED_NORMAL)。O(1)在静态优先级之外引入了动态优先级属性，并用它同时表示实时进程和非实时进程的优先级。进程的静态优先级是计算进程初始时间片的基础，动态优先级则决定了进程的实际调度优先顺序。无论是实时进程还是非实时进程，静态优先级都通过set_user_nice()来设置和改变，缺省值是120。

进程优先级的范围是[0,140)，实时进程的优先数，从1到99之间取值，非实时进程的优先数范围是[100,139]。优先级的计算主要由effect_prio()函数完成，计算公式如下：

$$prio=\begin{cases}100-1-rt\_priority & //实时进程的动态优先级\\ static\_priobonus-bonus & //非实时进程的动态优先级\end{cases}$$

其中，rt_priority是在setscheduler()中设置的一个常量值，所以实时进程的优先级一经设定就不再改变，非实时进程中的bonus计算公式为：

(NS_TO_JIFFIES ((p) ->sleep_avg)*MAX_BONUS/MAX_SLEEP_AVG)-MAX_BONUS/2,

由此可见非实时进程的优先级仅决定于静态优先级(static_prio)和进程的sleep_avg值两个因素，sleep_avg是进程的平均睡眠时间，是睡眠时间和执行时间的差值，进程休眠次数多、时间长，它们的sleep_avg也会相应地更大一些，所以计算出来的优先级也会相应高一些。

O(1)中动态优先级的计算不再集中在调度器选择候选进程的时候进行，只要进程状态发生改变，内核就有可能计算并设置进程的动态优先级，避免了系统中就绪进程计算过程耗时过长的问题。

2.4内核中的交互式进程优先策略实际效果有缺陷，2.6内核对交互式进程进行全面设计，提高交互式进程的优先权。在O(1)中通过以下认定便设为交互进程，需要优先考虑：

* 进程属性interactive_credit超过CREDIT_LIMIT阀值；

* 当进程时间片耗尽时，若宏TASK_INTERACTIVE()返回真，则不进入expired队列而保留在active队列以便尽快调度到这一交互式进程；

* 就绪等待时间升高优先级，超过一定阈值时。

2.6内核支持内核抢占，提高实时性能。无论是返回用户态还是返回核心态，都有可能检查NEED_RESCHED的状态；当返回核心态时，只要preempt_count为0，即当前进程目前允许抢占，就会根据NEED_RESCHED状态选择调用schedule()。在核心态中，因为时钟中断是不断发生的，因此，只要有进程设置了当前进程的NEED_RESCHED标志，当前进程就有可能马上被抢占，而无论它是否愿意放弃cpu，即使是核心进程也是如此。

2.6内核在多处理器间实现了负载平衡。当某个cpu负载过轻而另一个cpu负载较重时，系统会从负载重的cpu上"拉"进程过来，这个"拉"的负载平衡操作实现在load_balance()函数中。无论当前cpu是否繁忙或空闲，时钟中断每隔一段时间都会启动一次load_balance()平衡负载。为了减少进程在多处理器之间不断的"跳跃"，O(1)倾向于尽可能不做负载平衡，因此在判断是否应该"拉"的时候做了较多限制：

* 系统最繁忙cpu的负载超过当前cpu负载的25%时才进行负载平衡；

* 当前cpu负载取当前真实负载和上一次执行负载平衡时负载的较大值，平滑负载凹值；

* 各cpu负载取当前真实负载和上一次执行负载平衡时负载的较小值，平滑负载峰值；

* 对源、目的两个就绪队列加锁后再确认一次源就绪队列负载没有减小，否则取消负载平衡动作；

源就绪队列中以下三类进程参与负载情况计算，但不做实际迁移：(1)正在运行的进程；(2)不允许迁移到本cpu的进程；(3)进程所在cpu上一次调度事件发生的时间与进程被切换下来的时间之差小于某个阀值，说明该进程还比较活跃，cache可能还保存相关信息。

## 3 Linux 2.6基于公平策略的CFS调度处理过程分析

完全公平调度算法CFS从Linux 2.6.23后被采用，它从楼梯轮转终期调度算法RSDL/SD中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再区分交互式进程。而是将所有的进程都统一对待，确保在既定时间内，对于一定数量的可运行进程，每个进程获得公平的CPU占用。

内核仍采用schedule()函数进行调度，系统会根据已经注册的调度类，即通过函数task_tick_fair更新就绪队列，重新进行调度，从运行队列中选择下一个合适的进程运行：即红黑树中最左

技术创新

边的进程。

在 CFS 调度器中,采用了红黑树代替优先级数组。对于每个 CPU,CFS 的就绪队列(cfs_rq),使用按时间排序的红黑(red-black)树。红黑树是一种自平衡二叉搜索树,查找操作的时间复杂度为 O(logN),相比 O(1)调度器,CFS 具有可测量的延迟,但是对于较大的任务数无关紧要;红黑树可通过内部存储实现,即不需要使用外部分配即可对数据结构进行维护,节约了时间;与 O(1)相比,采用红黑树存储就绪队列节约了存储空间。

CFS 调度器用完全公平的策略代替动态优先级策略。完全公平的策略的关键就在于红黑树键值的计算方法。该键值由三个因子计算而得:一是进程已经占用的 CPU 时间;二是当前进程的 nice 值;三是当前的 cpu 负载。其中,进程已经占用的 CPU 时间对键值的影响最大,可以简单地认为键值就等于进程已占用的 CPU 时间,占有 CPU 的时间越长,键值越大,从而使得当前进程向红黑树的右侧移动。在 CFS 中,nice 值为 n 的进程比 nice 值为 n-1 的进程少获得 10%的 CPU 时间。例如:有两个任务 A、B:A 的 nice 值是 0,B 的 nice 值是 1,按照 CFS 的规定,A 应该会获得 55%的利用率,儿 B 会获得 45%的利用率。

技术创新

在计算键值时也考虑到这个因素,因此 nice 值越大,键值也越大。Linux 2.6.24 新增了调度周期 sched_period,引入了变量调度实体时钟 vruntime,即进程的执行时间,也即红黑树的键值(新创建的进程初始化值为 1,进入睡眠再唤醒时会做调整),在调度时,它先选择执行时间短的进程,se(调度实体)都是存在红黑树中的,也即 se->vruntime 越高,就越靠近 rb_tree 的右边,而 vruntime 小的 se 都会位于左边。sched_period 表示了所有就绪进程总的执行时间,每个任务将运行与其权重成比例的时间量。

当一个新任务变为可运行状态时,对其排队位置有严格的要求,在所有其他任务运行之前,此任务不能运行;否则,将破坏对以前任务作出的承诺。然而,由于该任务确实进行了排队,对运行队列的额外权重将缩短其他所有任务的时间片,在 sched_priod 的末尾释放一点位置,放置这个新的任务。

CFS 实现了模块化管理。为了支持实时进程,CFS 提供了调度器模块管理器。各种不同的调度器算法都可以作为一个模块注册到该管理器中。不同的进程可以选择使用不同的调度器模块。CFS 实现了两个调度模块,CFS 算法模块和实时调度模块。实时进程使用实时调度模块。CFS 调度模块用于以下调度策略:SCHED_NORMAL、SCHED_BATCH 和 SCHED_IDLE。CFS 的模块化是添加内核代码实现的。为使调度策略模块化,引入了调度类 sched_class,显著增强了内核调度程序的可扩展性。sched_class 封装了调度策略的接口函数,包括任务进入可运行状态 enqueue_task,任务退出可运行状态 dequeue_task,先出队后入队 yield_task,检查当前运行等的任务是否被抢占 check_preempt_curr 等。

Linux2.6.23 是基于单个进程进行公平调度,如果有进程 A,B,C,D,在相同优先级情况下,每个进程可以获得 25%的 cpu 时间。如果这四个进程中,只有进程 A 属于 user1 用户,而 B,C,D 属于 user2 用户。就造成了用户 user1 只占用了 25%的 cpu 时间,而 user2 却占据了 75%的 cpu 时间。在多用户系统上,这样的调度算法并不是用户所期望的。CFS 组调度更好地满足了用户要求,更好地支持服务器和台式机调度,改进了启发式处理。

选择组任务调度时和单任务一样,如在图 1 中按层次选取操作,若最高层两个组 A、B,两个进程 c、d,组 A 中有二个进程 a1, a2, 组 B 中有二个进程 b1,b2。在最顶层选取 se,也即在组 A、组 B,进程 c、进程 d 中选择,若选择的进程是 c,则调度这个进程。若选择的是组 A,则到组 A 中选,从 a1, a2 中选出一个合适的进程。

CFS 进一步提高了实时性。CFS 的调度时机还增加了在时钟中断中进行的检查,对处理进行了修改:当时钟中断发生时系统会调用函数 scheduler_tick (), 该函数会根据当前进程的 sched_class 调用 task_tick_fair 函数,更新运行队列信息,然后查看当前的进程是否为红黑树最左边的节点、若不是则抢占当前进程,否则当前进程继续占有 CPU。

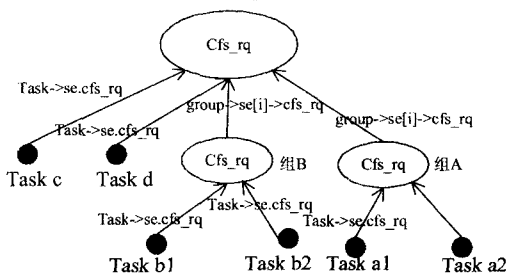


图 1 分层选取合适的任务

## 4 测试与适宜应用环境的分析

从以上 Linux 2.6 内核中采用的 CFS 和 O(1)两种调度处理过程的分析来看,两个算法从根本上来说是不同的,特别是在运行队列的存取和管理方面。但是两个算法都有一个共同目标:尽可能地实现公平调度,都利用了优先级和提高了 cpu 的利用率。

本文对 CFS 和 O(1)两种调度算法做了测试,从 nice 值的设定大小、进程占有 cpu 的时间段、进程对 cpu 的占用百分比、进程切换时的速度等几个指标衡量,结果显示两个调度算法都有很好的性能。但是,O(1)调度策略由于采用了优先级队列和查找位图,其挑选进程和插入进程的速度更快,因此更加适合实时进程调度时使用,而 CFS 调度策略由于红黑树的动态特性,在处理批处理进程和交互式进程时会更有优势。图 2 和图 3 显示了在特定 nice 值下 CPU 资源利用情况的测试结果。

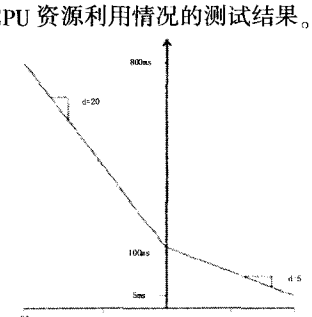


图 2 O(1)中 nice 值和占用 cpu 时间段关系图

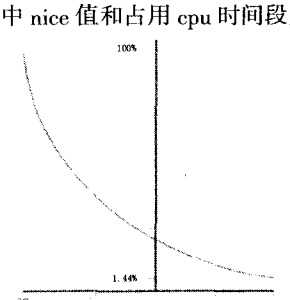


图 3 CFS 中 nice 值和占用 cpu 占用率关系图

(下转第 71 页)

实验说明我们的流量识别方案是合理的，而且识别效果好可以投入实际应用。

为了检验该多重识别算法的识别率是否优于单一的端口识别和应用层协议识别，我们来做个粗略的实验。在此引入指标PDI(数据识别率): PDI=D1/D 其中,D1 表示识别为 P2P 类型的数据量;D 为用于识别的全部数据量。在一个工作组中对系统进行了测试,采集了工作组与互联网之间的网络流量,此网络流量包括了 P2P 流量和一些非 P2P 流量共计 1243412KB 作为识别样本。分别使用端口识别、应用层协议识别和多重特性识别 3 种方式进行了识别,结果如表 2 所示。

表 3 数据识别率对比

| | 端口识别 | 应用层协议识别 | 多重特征识别 |
|---|---|---|---|
| 检出数据/KB | 17408 | 177311 | 461928 |
| PDI(%) | 1.4 | 14.26 | 37.15 |
| 总数据量/KB | | 1243412K | |

可以看出:端口识别的识别率最低,PDI 为 1.4%,说明多数 P2P 程序已不再使用固定不变的端口进行数据传输; 应用层协议识别的识别率相对高一些,PDI 为 14.26%;与应用层协议识别相比,多重特征识别的 PDI 高出 2 倍多,达到了 37.15%,本实验只是粗略的表明了多重识别算法能够具有比较高的 P2P 识别率,因为它能对加密的,未知的协议类型的 P2P 应用进行识别。

## 4 结束语

本文在研究了国内外现有 P2P 流量检测方法的基础上,针对现有 P2P 应用普遍采用随机端口,协议加密等技术带来的检测难题,提出了一种全新的 P2P 流量检测算法,该算法充分利用了 P2P 流量在传输层表现出来的行为特性同时借鉴了端口识别,DIP 识别的研究成果。实验表明该算法可以提高对 P2P 流量的识别能力, 同时还能够解决由端口跳跃和数据加密等引起的识别难题,具有比较好的实用价值。

本文创新点:提出了一种新的 P2P 流量检测方法,实验证明该方法可以提高对 P2P 流量的识别能力,同时还能识别端口跳跃和数据加密的 P2P 流,具有比较好的实用价值。

参考文献

[1]S Sen, J Wang. Analyzing peer-to-peer traffic across large networks [C]. The 2nd ACM SIGCOMM Workshop on Internet Measurement, Marseille, France, 2002

[2]SEN S, SPATSCHECK O, WANG Dong-mei. Accurate,scalable in-network identification of P2P traffic using application signatures [C]//Proceedings of the 13tthInternational Conference on World Wide Web. New York: ACM, 2004:512-521.

[3]KARAGIANNIS T,PAPAGIANNAKI K,FALOUTSOS M.Blinc: *multilevel traffit classification in the dark* [J1.SIGCOMM ComputerCommunications Review,2005,35(4):229—240.

[4]T Karagiannis, A Broido, M Faloutsos, et al. Transport layer identification of P2P t raffic [C]. The 4t h ACM SIGCOMM Conf on Internet Measurement, Taormina, Sicily, Italy, 2004

[5]Guanghui He, Jennifer Hou, et al. One Size Does Not Fit All: A Detailed Analysis and Modeling of P2P Traffic [C]//Proceedings in the IEEE GLOBECOM 2007: IEEE Computer Society, 2007: 393-398

[6]徐鹏,刘琼,林森 .改进的对等网络流量传输层识别方法[J].计算机研究与发展,2008,45 (5) : 794-802

[7]戴强 ,张宏丽.基于行为特征的 P2P 流量快速识别[J].微计算机信息,2009,1-3:28-33.

**作者简介:**刘剑刚(1984.9-),男,汉,硕士,湖南大学软件学院。专业:软件工程,研究方向:网络与信息安全; 秦拯,男,汉,博士,湖南大学软件学院教授。

**Biography:**LIU Jian-gang (1984.9-), male, Master, Software School of Hunan University. Major, Software Engineering; Research area, network and information security.

**(410082 长沙 湖南大学软件学院) 刘剑刚 秦 拯**

**(410082 长沙 湖南大学计算机与通信学院) 祝仰金**

**(School of Software, Changsha 410082, China) LIU Jian-gang QIN Zheng**

**(School of Computer and Communication, Hunan University, Changsha 410082, China) ZHU Yang-jin**

**通讯地址:(410082 长沙 湖南大学软件学院新软件大楼 306 室) 刘剑刚**

(收稿日期:2010.03.15)(修稿日期:2010.06.15)

(上接第 178 页)

## 5 结束语

通过以上对 Linux 2.6 内核调度中基于公平策略 O (1)和 CFS 的比较和分析表明,Linux 2.6 内核采用的调度更复杂且更有效率,2.6 各版内核先后采用的这些新的调度策略大大改进了 2.4 内核调度性能,选择 O(1)调度策略可应用于实时进程调度,选择 CFS 调度策略可应用于处理批处理进程和交互式进程,在不同的应用环境时应选择内核中不同的调度策略以进行相应适配。

参考文献

[1]毛德操,胡希明.Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001.pp263-414.

[2]於时才,廖东升等.Linux 2.6 调度系统的分析与改进[J].微计算机信息, 2007.24 卷 5-3 期, pp252-254.

[3]叶超, 郭立红, 邹荣士. Linux2.4 与 Linux2.6 内核调度器的比较研究[J].电子技术应用,2006,5:pp20-22.

[4]高博. linux 内核调度器分析及模拟.浙江大学硕士学位论文, 2008.5, pp5-21

[5]linux 内核源代码,v2.6.23[EB/OL].www.kernel.org,2007.7.12.

[6]linux 内核源代码,v2.6.10[EB/OL].www.kernel.org,2004.12.24.

[7]linux 内核源代码,v2.6.29[EB/OL].www.kernel.org,2009.7.2.

[8]linux 内核源代码,v2.6.24[EB/OL].www.kernel.org,2008.1.25.

**作者介绍:**李建萍(1983-),女,汉族,籍贯河南新乡,苏州大学计算机科学与技术学院,主要研究方向:计算机网络与信息安全。

**Biography:**LI Jian-ping (1983-), female, Xinxiang of Hen nan Province, Soochow University School of Computer Science and Technology, Research area:Computer Network and Information Security

**(215006 江苏 苏州大学计算机科学与技术学院) 李建萍 陆建德**

**(School of Computer Science and Technology of Soochow University, Suzhou, Jiangsu 215006, China) LI Jian-ping LU Jian-de**

**通讯地址: (215006 苏州大学计算机科学与技术学院) 李建萍**

(收稿日期:2010.03.15)(修稿日期:2010.06.15)