调度算法（一） - InuyashaSAMA - 博客园

InuyashaSAMA 关注 - *0* 粉丝 - *2*

- 前言
- 基本问题建模
  - 单机环境
  - 复杂环境：并行多处理机与工厂模型
- 基于优先级的贪心策略
  - 单机调度模式
    - 平均带权完成时间：$1||\sum w_jC_j$
    - 最大延时：$1||L_{max}$
    - 抢占式调度与发布时间：$1|r_j,pmtn|f$
  - 双机流水线模式
  - 并行处理机模式
    - 抢占式：$P|pmtn|C_{max}$
      - McNaughton’s wrap-around rule
    - 非抢占式：$P||C_{max}$
      - List Scheduling
      - Longest Processing Time First
      - 带依赖的任务：$P|prec|C_{max}$
      - LS for $O||C_{max}$
  - 优先级策略的限制
- 更精妙的策略
  - $1||f_{max}$的通用贪心解
    - Least Cost Last
    - 扩展到$1|prec|f_{max}$
    - 另一种解法
  - $1||\sum w_jU_j$的动态规划解
  - $P||C_{max}$的动态规划解
- 图与线性规划
  - 二分图匹配
    - $R||\sum C_j$问题
    - $O|pmtn|C_{max}$问题

前言

几个简单场景：

1. 一个CPU需要处理不断到达的程序。如何安排程序处理的顺序，最小化程序的平均处理时间（任务到达至完成的时间）。

2. 考虑一队五个宇航员准备重返太空。有一些任务需要在出发前完成。每个任务必须被分配给一个宇航员，且有的任务需要在其他任务完成后才能开始。如何安排任务分配，使得所有任务的完成时间最小。

3. 考虑一个生产不同类型产品的工厂。不同产品需要不同机器上的不同处理时间，需要首先在机器1处理，然后是机器2，最后机器3，不同产品在不同机器上处理时间不同。工厂会收到订单，每个订单有额定的完成时间，必须在此之前完成。如何安排机器的生产顺序，使得工厂完成尽可能多的订单。

更一般地说，调度问题通常指一系列任务需要分配到一些机器上，满足某些约束，并且优化一个特定的目标函数。从各种各样的应用课题中可以挖掘出成千上万的调度问题建模，因此几乎不可能完全覆盖这些问题。因此，我们将从一些基本问题出发，介绍一些对解决调度问题有用的算法设计技术。

我们仅聚焦于多项式时间算法，但是实际上许多这类问题都是NP难问题，显然很难有确定性的多项式时间算法来得出最优解。在这些问题中，我们通常对相关的近似算法感兴趣。

基本问题建模

一个调度问题通常有三个元素：机器环境，最优条件，其余边界约束。我们首先从最简单的机器环境开始，然后再讨论更复杂的。

单机环境

假设有n个任务的任务集合J，在单机环境中，有一个在同一时间只能处理单个任务的处理机。任务j需要的处理时间为p_j，如果一个任务一旦开始就必须直接执行完，则称该调度环境是非抢占式的，否则是抢占式调度。对任务集J的一个调度S，指定机器何时调出p_j的时间来执行任务j，任务j在调度S中的完成时间记为C_j^S。

一个调度算法的目标是计算一个“好”的调度策略，但是“好”的定义取决于不同的应用。必须要确定一个最优性条件，调度算法的最终目标是给出满足该条件的调度策略。下面考虑一些边界约束，每个任务j具有发布时间r_j，只有在任务发布后才能被处理。同时任务之间还有偏序$<$，只有所有满足$j' < j$的任务j'被完成后，j才能被处理。

现在可以考虑一个简单的问题模型$\alpha|\beta|\gamma$，在单机问题中，$\alpha = 1$代表单机环境，β指边界约束，γ指优化目标。如果要优化所有任务的平均完成时间，则$\gamma = \sum C_j$，若是追求所有任务的完成时间，$\gamma = C_{max}$。在当前环境下，β是$r_j, prec, pmtn$的子集，分别代表发布时间，任务间的偏序约束，是否抢占式调度。上面提到的场景1就可以用$1||\sum C_j$来建模。

此外，还有两个其他可能的元素会影响单机调度环境。

- 任务有不同的优先级，某些任务需要被优先处理，因此为任务j分配一个可能的权重w_j，权重越大，其完成时间相关惩罚就越大，因此$\gamma = \sum w_j C_j$。
- 任务j有一个截止时间d_j，规定它应该被完成的时间，这会导致两种不同的优化函数。对于调度策略S，定义$L_j = C_j^S - d_j$为任务延时，我们可以选择最小化n个任务的最大延时：$\gamma = L_{max}$。也可以选择最大化在截止时间之前完成的任务的数量，据此可以定义$U_j = 0$若$C_j^S \le d_j$，否则$U_j = 1$。此时可以定义$\gamma = \sum U_j$，当然也可以加上权重$\sum w_j U_j$。

最后，我们考虑一个更一般的优化条件。对任务j，定义$f_j(t)$为随完成时间t的非减函数。对调度策略S，定义$f_{max} = \max_{j=1}^n f_j(C_j^S)$。后面会考虑到的一个问题建模就是$1|prec|f_{max}$。

复杂环境：并行多处理机与工厂模型

现在我们考虑一些更复杂的环境，首先是并行机器环境。假设有m个机器，任务j可被任意一个机器处理，当然如果允许抢占式调度，那么一个任务可以先在一个机器上处理，再转移到另一个。单个机器同时只能处理不多于一个任务，单个任务同时只能在不多于一个机器上运行。

在等价（identical）并行环境中，每个处理机完全相等。在均匀相关（uniformly related）并行环境中，每个机器i具有速度s_i，因此任务j在机器i上的运行时间将是p_j/s_i。在不相关（unrelated）环境中，每个机器的速度还和任务相关，即s_{ij}，表示机器i运行任务j的速度，此时运行时间将为$p_{ij} = p_j/s_{ij}$。

在工厂环境中，也有m个机器，但是任务由不同的操作组成，每一个操作需要在特定的机器上完成，不同操作可能需要不同的时间来完成。在开放工厂（open shop）中，每个商品的不同处理操作可以以任意顺序完成，只要保证两个操作不是同时在不同机器上处理的。而在任务工厂（job shop）中，不同的操作之间具有一个全序关系，一个操作必须在其所有前驱操作完成后才被执行。任务工厂的一个特殊例子是流水线工厂（flow shop），它还要求不同任务在机器上的处理时间不同。在流工厂和开放工厂中，每个任务都需要在每个机器上处理一次，完成特定操作。

上面介绍的等价并行，均匀相关，不相关环境分别用P, Q和R来表示，开放工厂，流水线工厂，任务工厂分别用O,F,J表示。至此前面提到的简单场景2可以用$P5|prec|C_{max}$建模，场景3可用$F3|r_j|\sum U_j$建模。

基于优先级的贪心策略

最显然的解决上述问题的调度算法就是贪心选择：每当一个机器空闲下来，就分配给它一个任务。我们把算法再稍微设计的精致一点的话，就为每个任务定义一个优先级函数（根据最优条件γ），然后分配时分配最高优先级的任务。在这一章节，我们讨论单机环境，多机并行环境，工厂环境下如何设计这样的贪心策略。在设计的时候，优先级函数仅与当前任务j相关，这样的话调度算法仅需要两步：计算优先级，排序。显然时间复杂度为$O(n\log n)$。我们也将探讨这样设计的一些缺陷。

单机调度模式

首先关注单机调度问题下的优先级策略：按照优先级对任务进行排序，而后以此顺序作为调度策略。如果要证明此类算法的最优性，通常采用交换的方法来论证：如果存在一个最优策略，其执行顺序不符合按照优先级排序的结果，也就说明这个顺序中存在两个任务顺序与优先级不符，我们证明交换这两个任务能带来更好的结果，以此说明这样的最优策略不存在（也就是常见的反证）。

平均带权完成时间：$1||\sum w_jC_j$

这大概是最简单的模型了，优化所有任务的完成时间之和$\sum C_j$。直觉上看，我们应该把花费时间最长的任务放在最后做，这样其花费就不会累加在其余任务之上，这就是最短任务优先算法（shortest processing time）：按处理时间p_i非减地排序所有任务，按照这个顺序来调度任务。

定理3.1.1：SPT算法可以最优的解决$1||\sum C_j$调度问题。

证明：假设存在两个任务j和k按照相邻顺序参与调度，但是$p_j > p_k$，形成了一个最优调度。现在我们将两个任务对调，此时由于j和k相邻，其他任务的完成时间不变，唯一变化的是j和k，假设两个任务开始于时间t，则对调前，两个任务的完成时间之和为$(t + p_j) + (t + p_j + p_k)$，对调后，完成时间为$(t + p_k) + (t + p_k + p_j)$，变化为$p_k - p_j$

，由于$p_j > p_k$，因此对调后完成时间更优，与假设的最优性矛盾。

实际上对于带权的问题$1||\sum w_j C_j$也可以按照这个思路来证明，直觉上我们要充分利用每一段单位时间的权重，单位时间内获得的权重越大越好，也就是按照w_j/p_j升序排列。其证明可以简单地由上面的证明推演出。

最大延时：$1||L_{max}$

一个很自然的想法是，首先调度最紧迫的任务，这催生出最早截止算法（earliest due date）EDD：按任务的截止时间升序排列并调度之。可以证明，EDD对于$1||L_{max}$是最优的。

定理3.1.2：EDD算法对于$1||L_{max}$是最优的。

证明：同样适用交换论证法。不失一般性，假设所有任务的截止时间不重复，且$d_1 < d_2 < ... < d_n$。在所有可能的最优调度中选择具有最少逆序的一个调度，其中有一些任务对j和k，$j < k$但是调度顺序相反，显然它不是EDD调度。假设j和k是相邻的（必然存在这样的任务对），我们替换两者不改变其他任务的完成时间，同样也不改变其延时。唯二变化的就是这两个任务，下面证明的目标是交换后降低了$\max(L_j, L_k)$，不改变调度的最优性。由于在替换前，j后于k调度，意味着$C_j^S > C_k^S$，但是$d_j < d_k$，故$\max(L_j, L_k) = C_j^S - d_j$，替换后任务$j$的完成时间减少了，但是任务$k$的上升到原来的$C_j^S$，因此两者之间最大延时不大于$C_j^S - d_k$，这小于替换前的结果。这样的话，由于交换了$j$和$k$，我们减少了这个调度中的逆序，但是它依然至少是最优的，这和假设矛盾。

抢占式调度与发布时间：$1|r_j, pmtn|f$

现在考虑更复杂点的单机调度，不同的任务会在不同的时间到达，即发布时间r_j，显然上面提到的贪心策略不能直接使用，因为优先级高的任务不一定会早于优先级低的任务被发布。要处理这样的情况，最直接的方法就是不管他，每次调度都处理**当前已发布**的任务中优先级最高的任务。在抢占式调度中，这意味着一旦一个更高优先级的任务发布，当前正在执行的任务就要挂起，把处理器资源让出来。我们将会证明这个策略在抢占式调度中是最优的。

定义最短剩余时间优先算法（shortest remaining processing time, $SRPT$）：每个时间点按照最短剩余时间调度，当有剩余时间小于当前任务时，抢占处理机。同时定义新的EDD算法：按照当前已发布的最早截止时间任务抢占式调度。可以证明，$SRPT$是调度$1|r_j, pmtn|\sum C_j$问题的最优算法，EDD也是$1|r_j, pmtn|L_{max}$的最优解。

证明：和之前一样采用反证法导出矛盾，不同的是我们这次交换的是不同任务的片段（考虑到抢占式调度）。首先我们考虑$1|r_j, pmtn|\sum C_j$问题，假设一个最优调度中具有最短剩余时间的已发布任务j没有在t时刻被调度，而是任务k被调度，且剩余时间$p'_j < p'_k$。总的来说，时间$p'_j + p'_k$将在t时刻之后被占用。现在进行一次替换：将$p'_j + p'_k$的前p'_j时间片用于执行任务j而不是任务k，剩余时间用于任务k。在新的调度策略中，j和k以外的任务完成时间均不变，但是对于j和k，由于j剩余时间小，替换后$C_j + C_k$降低了，矛盾，从而$SRPT$是最优的。对于EDD的证明如出一辙，将前面的时间片用于截止时间更早的任务j后，不增加最大延迟（分析方法和之前类似）。

之前考虑的都是抢占式调度下，具有发布时间R的调度策略，分别是$SRPT$和EDD。但是对于非抢占式调度，由于不能简单地挂起当前任务执行新任务，我们必须决定要不要停机等待一个更高优先级任务，还是直接执行面前的低优先级任务，这在直觉上很难计算最优解。实际上，$1|r_j|\sum C_j$和$1|r_j|L_{max}$都已被证明为NP难问题，我们将在后续介绍相关的近似算法。

同样地，上面的贪心策略也不适用于带权重的任务调度，$1|r_j,pmtm|\sum w_jC_j$也被证明为NPH问题。同时$SRPT$和EDD称为**在线调度**算法，因为它们不依赖于当前时刻未知的条件，仅依赖当前已发布的任务的优先级来进行调度。文献[38]是一个完整的在线调度综述。

双机流水线模式

另一个更复杂的问题是在流水线工厂中最小化任务的最终完成时间，一般来说，对于大于等于3个机器，这个问题是NP难的。但是特殊情况，对于双机流水线$F2||C_{max}$，存在一个基于优先级的最优算法。记(a_j,b_j)为任务j需要在两个机器上进行处理的时间。直觉上，我们应该让第一个机器尽可能快地处理短任务，以让第二个机器减少等待；同时应该让第二个机器先处理长任务，因为这些任务本身就代表着较大的完成时间，缩小它们有助于缩小最终的C_{max}。

下面将这个想法形式化：将所有任务分成两个集合，A表示$a_j \le b_j$的任务集合，B表示$a_j > b_j$的任务集合。定义一个$Johnson$规则：首先将A集合中的任务按照a_j升序排列，再将B集合中的任务按照b_j降序排列，按这个顺序调度两个机器上的所有任务。

注意：我们并没有在第二个机器上重排任务，所有任务的两个操作在两个处理机上都是按相同顺序执行，这种调度称为**排列调度**（permutation schedule）。注意：超过2个机器就不一定有最优排列调度了，可能需要在第二个机器上重排。

定理3.2.1：$F2||C_{max}$问题的实例总是存在一个最优的排列调度。

证明：考虑任意一个最优调度S，按照任务在机器1上完成的时间对任务进行编号。假设存在任务对$j<k$（这意味着在机器1上j先于k完成），在机器2上，任务j紧接着k被执行，并且k在机器2上开始执行的时间为t，这说明任务j在机器1上完成的时间早于t，在时间t时，两个任务都可以在机器2上执行。因此我们替换任务j和k在机器2上的执行顺序，这不改变最终完成时间。因此可以不断进行这样的替换并保持最优性，直到不存在满足条件的任务对，最终所有任务在机器2上的执行顺序等同于机器1，我们就将这个最优调度替换成了排列调度，证毕。

现在我们可以将问题的解空间限制在排列调度上了（因为存在最优的排列调度）。

定理3.2.2：通过$Johnson$规则形成的任务排列是双机流水线模式的最优调度。

证明：注意到，在一个排列调度中，必然存在一个任务k在机器1的操作完成后可以不经过等待直接开始在机器2上的操作。因此n个任务的完成时间包含前k个任务在机器1上的处理时间和$n-k+1$个任务在机器2上的处理时间。这些加起来一共包含$n+1$个任务的处理时间（$n+1$个a_i或b_j的组合），因此如果把所有的a_i和b_i减去相同的时间p，那么在排列调度下，最终完成时间将减少$(n+1)p$。

同时，还注意到，如果一个任务的a_i等于0，它必然在某个最优排列调度中被最优先调度，因为它不占用机器1的时间，反而可以填充机器2可能的空闲等待时间，充分利用机器2。类似的，如果一个任务的b_i等于0，它必然在某些最优排列中被放在最后执行（因为机器1是无空闲的，而在机器2上不占时间，不能提高机器2的利用率）。

因此，我们可以这样构造一个最优的排列调度：不断寻找未调度任务中具有最小a_j或b_j的任务j，然后将所有未调度任务的处理时间减去这个最小值，此时任务j的a_j或b_j为0，按照上面的分析将其置于排列的最前或最后。

容易看出，这样的构造产生的最优排列满足$Johnson$规则。

并行处理机模式

现在我们考虑等价并行环境P，机器一旦多起来，很多原先在单机环境下很简单的问题变成了NP难问题。因此我们倾向于寻找问题的近似算法。实际上，在一些情况下，单机环境下简单的优先级调度也能够在并行环境下得到不错的效果。这类算法的通常步骤是：每当一台机器变得空闲，就把当前优先级最高的任务分配给它。这种不放任任何机器空闲的调度策略被称为“繁忙”调度（busy schedule）。

在这一节，我们还将介绍一个新的分析方法。我们将给出一个最优调度策略质量的下界，而不再使用交换法论证策略最优性。而后我们将论证提出的近似算法能够达到最优策略下界的附近，比如差一个因子。这是分析近似算法的常见技术，保证了近似算法的解能够以比例逼近最优解，虽然我们并不知道最优解的位置。有时我们甚至能论证新的贪心算法能够到达最优的下界，这就意味着该贪心算法实际上是最优的。

首先聚焦于最小化m个并行机上的平均完成时间，即$P_m||\sum C_j$。在这个问题下，贪心的SPT算法依然是最优的。后面会进行证明。

而后考虑最小化最终完成时间C_{max}，在单机环境下，$1||C_{max}$基本不是一个问题，只要保证机器不空闲，随便怎么调度结果都一样。但是机器一旦多起来，问题就很复杂了。如果允许抢占式调度，存在一个最优的多项式时间的贪心算法。但是在非抢占式条件下就不行了，它已被证明是NPC问题[7]，我们将给出一个它的近似算法，首先我们证明任何繁忙调度是2-近似的，然后我们介绍一个更聪明的4/3-近似算法——LPT算法，最后我们将给出一个更优但也更复杂的算法。

我们对这些算法的分析是基于比较它的解与最优解下界的差距，因此和最优解的差距会更小。下面是两个简单的C_{max}的下界：

第一个下界说明，最优调度策略得到的完成时间至少是每个机器的平均负载，第二个下界说明，最终完成时间至少是任何一个任务的完成时间。这两个下界都是很显然的，且对于抢占式和非抢占式都有效。首先对于抢占式问题$P|pmtn|C_{max}$，我们将给出一个最优调度算法，它能够达到上面两个下界的最大值；后续对于非抢占式调度，我们将使用这个下界来确定近似解。

抢占式：$P|pmtn|C_{max}$

McNaughton’s wrap-around rule

$McNaughton's$规则是一个简单的构造$P|pmtn|C_{max}$问题最优解的方法，它最多需要$m-1$次抢占。这个算法和许多其他调度算法不同，以机器为单位调度任务，而非时间。

定理3.3.1：用$McNaughton's$规则构造$P|pmtn|C_{max}$的策略，其最终完成时间达到最优下界。

证明：首先定义D是最优下界，它等于上述的两个下界的最大值：$D=max\{\sum_j p_j/m, \max_j p_j\}$。而后，我们准备把任务片段分配给各个机器：按顺序从机器1至m，任务1至j进行分配，每个机器分配的处理时间都不超过D，机器i分配满D时间后才能开始机器$i+1$的分配。每个任务j有最后的t时间片在机器i上执行，前p_j-t时间片在机器$i+1$上运行。由于D不小于单个任务的处理时间，因此一个任务最多被抢占一次分在两个机器上执行。又由于mD不小于所有任务的总处理时间，按照这种方法分配，所有任务都一定能够被分配成功，这说明会有一个任务不需要抢占（否则需要$m+1$个机器），因此该算法共抢占$m-1$次。

非抢占式：$P||C_{max}$

$P||C_{max}$是已证明的NP难问题，我们给出几个简单的近似算法。

List Scheduling

这是一个极其简单的贪心策略：每当有新的机器空闲，就为其分配任意一个未处理的任务，但它竟然是2-近似的。

定理3.3.2：LS算法是$P||C_{max}$的2-近似算法。

证明：假设任务j是在一次LS调度中最后完成的任务，那么它的完成时间就是最终完成时间C_{max}，设它的处理时间为p_j，开始时间为s_j，则$C_{max} = s_j + p_j$。注意到在时间点s_j之前，所有的机器应该都是繁忙的，否则任务j将更早开始。而所有机器同时保持繁忙的时间不会超过$\sum_{j=1}^{n} p_j/m$，否则的话所有任务都完成了。因此我们有：

$$C_{max}^{LS} \leq s_j + p_j \leq \sum_{j=1}^{n} p_j/m + p_j \leq 2C_{max}^{*}$$

这个算法很简单，其得到的完成时间不超过最优解的两倍，实际上即使任务有发布时间r_j，即对于$P|r_j|C_{max}$，这个算法也是2-近似的，证明方法类似，就不赘述了。

Longest Processing Time First

在上面的LS算法中，我们每次分配任务都是任意选择的，我们在分析的时候选择的瓶颈值是最后一个处理完成的任务，因此其完成时间不超过$\sum_{j=1}^{n} p_j/m + p_j$，一个很自然的想法是，让最后一个完成的任务尽可能短，这样完成时间的上界是不是就更接近最优呢？这就是最长处理时间优先调度LPT：在LS调度的基础上，每次选择处理时间最长的任务进行分配。

定理3.3.3：LPT算法是$P||C_{max}$的$4/3$-近似算法。

证明：依然选取最后完成的任务j，它开始于时间点s_j，但它并不一定是最短任务，因为可能有其他任务在s_j之后开始但是在$s_j + p_j$之前完成。因此，我们把所有在s_j之后开始的任务移除，这不改变最终完成时间。只要对移除后的问题实例分析即可，在新的问题中，任务j同时也是最后开始的任务，这说明$p_j = p_{min}$。

参照LS算法的分析，LPT的调度结果$C_{max}^{LPT} \leq C_{max}^{*} + p_{min}$，如果$p_{min} \leq C_{max}^{*}/3$，不证自明，因此只需证明$p_{min} > C_{max}^{*}/3$的情形下，$LPT$就是最优调度：

在这种情况下，任意的$p_j > C_{max}^{*}/3$，意味着最优调度中，每个机器分配的任务数不超过2，假设有m个机器，n个任务，这说明$n \leq 2m$，当$n \leq m$，显然最优调度就是每个机器分配一个任务，这和LPT是一致的，因此考虑另一种情况$m < n \leq 2m$，在这种情况下，LPT调度是将任务按照处理时间降序排列，前m个任务按顺序分配给m个机器，后续任务k将与$2m+1-k$配对在同一机器上。考虑一个非LPT的最优调度，存在机器i和机器j均被分配了2个任务，假设先分配了p_1和p_2，其中$p_1 > p_2$，后续分配的任务p_i与p_j分别于其配对，但是$p_i > p_j$，不符合LPT调度，我们可以简单地替换i和j来降低两个机器的完成时间，产生矛盾。因此LPT调度就是这种情况下的最优解。

带依赖的任务：$P|prec|C_{max}$

当输入的任务存在依赖关系时，LS调度依然是非抢占式调度的一个选择，当然会有一些小修改。我们说一个任务在t时刻是**可运行**的，如果其所有前驱均已在t时刻被完成。在这种情况下，LS调度将每次选择任意**可运行**的任务调度给空闲机器。同时，假设任务集中存在的任意一条任务链为$j_{i_1} < j_{i_2} < ... < j_{i_k}$，则任意最优的调度策略的完成

时间必然不会小于任务链上任务的处理时长：

这是有依赖关系的任务调度中新增加的下界。

定理3.3.4：LS算法是$P|prec|C_{max}$的2-近似算法。

证明：假设j_1是调度策略中最晚完成的任务，定义j_2是j_1的前驱任务中最晚完成的任务，如此递归地定义j_l，直到任务j_k不存在前驱，构成集合$C=\{j_1,j_2...,j_k\}$，我们把LS调度策略的时间消耗分成两部分，A表示所有C中任务正在某个机器上运行的时间点，B表示其余的时间点，则$C_{max}=|A|+|B|$。注意到：B中所有时间点上，m个机器必然都是繁忙状态，否则，如果有机器处于空闲状态，而此时必然存在C中可运行的任务（C中没有任务在运行），则该机器应当直接运行该任务。因此根据前面的分析，所有机器同时保持繁忙的时间不会超过$\sum_{j=1}^{n}p_j/m$，从而$C_{max}=|A|+|B|\le\sum_{j\in C}p_j$ $+\sum_{j=1}^{n}p_j/m\le 2C_{max}^*$。

LS for $O||C_{max}$

LS调度还可以应用于$O||C_{max}$问题，开放工厂问题下，每个任务需要在不相交的时间区间内在不同机器上进行处理。设P_{max}是单个任务在所有机器上的处理时间之和的最大值，Π_{max}指单个机器上所有任务的处理时间之和的最大值，显然两者均是最优调度的下界。

定理3.3.5：LS算法是$O||C_{max}$的2-近似算法。

证明：假设机器M是最后完成处理的机器，j是机器M上最后处理的任务。在任何时间点，要么机器M在处理中，要么任务j在某个机器上处理，否则任务j将在机器M上处理。j的总处理时间不超过P_{max}，而M的总处理时间不超过Π_{max}，因此LS调度中，总处理时间最多不超过$P_{max}+\Pi_{max}$，从而不超过$2C_{max}^*$。

优先级策略的限制

对许多问题，简单的调度策略并不能达到很好的效果，因此我们在设计算法的时候，必须小心地选择策略组合。实际上对于很多问题，特别是任务之间存在依赖关系，或是有发布时间的情况，最优调度通常允许机器有空闲时间。而应用贪心策略得到的调度通常是繁忙调度，往往导致次优解。

考虑$Q||C_{max}$问题的简单实例：两个任务，两个机器，机器1的运行速度为1，机器2的速度为x，其中$x>2$，两个任务的处理时间均为1。应用任何贪心调度策略，都会将两个任务分在两个机器上，完成时间为1。但很显然的是，如果我们将两个任务都分在机器2上，完成时间是$2/x$，更优。当x无限扩大时，没有任何贪心策略的解能够达到固定的近似比。实际上对于这个问题，我们还是能找到简单的2-近似启发式算法来解决问题，但是对于类似$R||C_{max}$和$Q|prec|C_{max}$这类复杂问题，就没有简单的近似算法。

更精妙的策略

基于优先级的调度算法把任务分开来单独看待，在很多问题上会错过最优解。在这一节，我们考虑更复杂的策略，同时考虑其他任务的信息来辅助决策，而不仅仅按照优先级排序。这些算法是递增迭代的，通常从一个空集开始，每次调度一个任务，直到达到最优解。每次任务的选择是根据已分配的任务上下文来计算的。我们将给出两个经典的动态规划算法，以及另外几个更特别的算法。

$1||f_{max}$的通用贪心解

我们解决的第一个问题是$1||f_{max}$，这是之前定义过的一大类问题，每个任务存在一个非减的惩罚函数$f_j(C_j)$，问题的目标是最小化所有任务的最大惩罚值，例如$1||L_{max}$中，$f_j(t)=t-d_j$。

Least Cost Last

我们定义$p(J)=\sum_{j\in J}p_j$为所有剩余任务的处理时间之和，J是未调度的任务集合，任何任务都在$p(J)$时间之前完成，算法的主体是找到使$f_j(p(J))$最小的任务j，将其放在最后执行，而后递归地对剩余任务应用上述步骤，这个算法称为Least-Cost-Last。

注意到此算法和我们之前的几个贪心算法的不同之处，之前的贪心策略可以按照同一个标准直接对n个任务排序，因此可以应用$O(n\log n)$的排序算法完成，但是此算法必须调度任务j后，才能根据新的$p(J)$值确定下一个被调度的任务，因此其时间复杂度为$O(n^2)$。此算法最优性的证明依然是通过最优解的下界来设计。

定义$f^*_{max}(J)$为任务集J的最优调度下的惩罚函数值，可以推导出两个下界：

第一个下界的来源是，必然有一个任务k最晚完成，它的惩罚函数参数恰好是$p(J)$，从而有$f^*_{max}(J)\geq f_k(p(J))\geq \min_{j\in J}f_j(p(J))$。第二个下界来自于这样一个事实：如果从任务集中去掉一个任务，不会增加任何任务的完成时间，而惩罚函数是完成时间的非减函数，这意味着新的任务集的最优调度惩罚不增。

定理4.1.1：LCL算法是$1||f_{max}$的最优解。

证明：根据算法步骤，我们首先选取使得$f_j(p(J))$最小的j放在最后，其惩罚为$\min_{j\in J}f_j(p(J))$，而后剩余任务集为$J-\{j\}$，在这个任务集上的最优解为$f^*_{max}(J-\{j\})$，因此LCL调度的惩罚值为$\max\{\min_{j\in J}f_j(p(J)),f^*_{max}(J-\{j\})\}$，恰好是前面导出的最优解的下界。

扩展到$1|prec|f_{max}$

即使引入任务间依赖，Least-Cost-Last算法依然是最优的。在$1|prec|f_{max}$问题中，由于任务间存在依赖，因此稍微修改一下算法描述：我们将从当前未调度的无后继任务集L中，选择使$f_j(p(J))$最小的任务。这样的话，可以推导出一个新的下界：

因此其最优性的证明完全类似上一节的证明，不再赘述。

另一种解法

$Moore$[33]给出过另一种$1||f_{max}$的解法，他的策略基于前面提到过的针对L_{max}的EDD算法。假设我们想知道是否存在一个调度能够使得$f_{max}\leq B$，可以给任务j分配一个截止时间d_j，使得$d_j=argmax_t\{f_j(t)\leq B\}$，易知，如果每个任务都能在截止时间之前完成，那么该调度的$f_{max}\leq B$等价于$L_{max}\leq 0$（否则只要有一个任务晚于截止时间，惩罚就超过B），因此我们将f_{max}问题扩充了截止时间后，转化为了最小化最大延迟问题。因此一个想法是：对B进行二分查找，然后构造相应的截止时间，应用EDD来计算L_{max}，从而逼近最优解。

$1||\sum w_jU_j$的动态规划解

现在我们考虑$1||\sum w_jU_j$问题，其优化目标是最小化已超时任务的总权重。此问题是弱NPC问题，即具有伪多项式时间算法，该算法复杂度为$O(n\sum w_j)$，也就是说如果任务权重和能被一个n的多项式约束，那么算法可以在多项式时间内结束。实际上，如果所有权重都为1，那么问题$1||\sum U_j$可以直接用该算法解决，且时间复杂度退化为$O(n^2)$。而且，这个算法还可以用来导出一个此问题的$(1+\epsilon)$-近似算法，该算法复杂度是关于n和$1/\epsilon$的多项式。

首先注意到，在这个问题中，任务集被划分为两种，一是在截止时间之前完成的任务，一是超时完成的任务。如果存在一个调度策略，使得一个任务集中所有任务都能在截止时间之前完成，则称这个任务集是**可行**的。显然，当一个任务集是可行的，那么其调度策略可以直接使用EDD，因为EDD最小化最大延迟，在可行任务集下必然存在最大延迟不超过0的调度，从而使所有任务能按时完成。调度完成后，该可行任务集的**完成时间**就是所有任务的处理时间之和$\sum p_j$。

现在我们考虑$1||\sum w_jU_j$问题，它实际上是在寻找任务集$\{1,2,...n\}$中权重和最大的**可行**子集。首先将所有任务按照截止时间升序排列并编号，定义T_{wj}为任务集$\{1,2,...j\}$的权重和不小于w的**可行**子集族中，具有最小**完成时间**的子集的完成时间。如果不存在任何可行子集，则记为∞。边界条件：记$T_{w0}=\infty$, $T_{0j}=0$。现在尝试计算$T_{w,j+1}$，考虑$\{1,2,...,j+1\}$中权重和不小于w的完成时间最小的可行子集S：

- 如果$j+1\in S$，则我们按照EDD对S进行调度，$j+1$应放在最后执行，$S-\{j+1\}$的权重和不小于$w-w_{j+1}$，且$S-\{j+1\}\in\{1,2...j\}$，故$T_{w,j+1}=T_{w-w_{j+1},j}+p_{j+1}$，当然如果计算出的$T_{w,j+1}$超过了截止时间$d_{j+1}$，$j+1$就不可以放在集合$S$中。
- 如果$j+1\notin S$，显然$T_{w,j+1}=T_{w,j}$。

状态方程总结如下：

$$T_{w,j+1}=\begin{cases}min(T_{w,j},T_{w-w_{j+1},j}+p_{j+1}) & if\ T_{w-w_{j+1},j}+p_{j+1}\le d_{j+1}\\ T_{w,j} & otherwise\end{cases}$$

显然，不存在权重超过$\sum w_j$的可行子集，因此一旦w超过该值就可以终止程序。覆盖所有w值的时间复杂度为$O(n\sum w_j)$，一旦计算完所有的T_{wj}，我们只需要找到使得$T_{wn}<\infty$的最大的w对应的可行子集，对该子集进行EDD调度，而后将其余任务最后调度，这就是$1||\sum w_jU_j$的最优调度策略。

$P||C_{max}$的动态规划解

我们前面已经介绍过一些$P||C_{max}$的近似算法，其中最长任务优先算法LPT能够达到4/3的近似比，同时我们也知道这个问题是NP难的。这一节我们假设所有任务的处理时间都在一个**有限集**中选择，则问题可以在多项式时间内解决。

定理4.2.1：$P||C_{max}$问题中，任务数n，机器数为m，p_j是任务j的处理时间，如果满足$\{p_1,p_2,...p_j\}\subseteq S$且$|S|=s\le\infty$，则存在一个最优调度算法，其时间复杂度不超过$n^{O(s)}$。

证明：同样适用动态规划的思路，设$S=\{z_1,z_2,...,z_s\}$，注意到在上面的约束下，一

台机器上运行的任务集可以用一个s维向量来表示：$v=(v_1,v_2,...,v_s)$，其中v_k表示在此机器上执行的处理时间为z_k的任务的数量。同时由于共有n个任务，意味着这样的s维向量共有n^s个。设T是目标完成时间，也就是说所有机器需要在T时刻之前完成所有分配的任务。设V是所有处理时间小于T的（即$\sum v_i z_i \le T$）的向量集合。在满足目标完成时间T的调度策略中，每个机器必然都从V中选择一个任务集来执行。定义$M(x_1,...,x_s)$是完成任务集$(x_1,x_2,...x_s)$的最小机器数量。容易看出下面的等式成立：

$$M(x_1,...,x_s)=1+\min_{v\in V} M(x_1-v_1,...,x_s-v_s).$$

实际上就是从中去除一个机器承担的任务后，递归地计算剩余任务所需的最小机器数，最终计算最小值。完整的执行这个动态规划算法需要首先遍历n^s个向量组合，每个向量需要遍历n^s次剩余任务，因此时间复杂度为$O(n^{2s})$。

还需要考虑的是T的选择，可以在所有可能的任务向量对应的处理时间中选择T，应用算法确定最小机器数目，如果数目大于机器数m则减小T，否则增加T，据此可以应用二分查找。最终得到使得$M=m$的最小的T对应的调度策略就是最优调度。

图与线性规划

网络图算法和线性规划算法是组合优化问题的核心主题。能够用来解决许多难题，当然也能用于调度算法的设计。在这一节，我们考虑应用二分图匹配和线性规划算法来解决调度问题。

二分图匹配

一个二分图包含两个点集A和B，以及边集$E\subseteq A\times B$，它的一个匹配M定义为E的子集，且任何一个顶点都至多在M中的一条边上。我们可以考虑将任务和机器进行匹配，这样的话根据上面的定义，每个任务最多只会被分配给一台机器，同时一台机器最多只会被分配一个任务。如果$|A|\le|B|$，则如果A中所有结点都在匹配M的一条边上，则称这个匹配是**完美匹配**。当然，也可以为每条边赋权重，而后定义匹配的权重为其中所有边的权重之和。一个很重要的事实是，二分图的最小完美匹配是有多项式时间算法的。

$R||\sum C_j$问题

在不相关并行环境下优化调度的平均完成时间，似乎是一个挺难的问题，本节将给出一个多项式时间算法。对任何调度，设κ_{ik}为在机器i上倒数第k个运行的任务，l_i为机器i上运行的任务数量。注意到，一个任务的完成时间等于在该任务之前（包括自身）运行的任务处理时间之和，我们有：

$$\sum_j C_j=\sum_{i=1}^{m}\sum_{k=1}^{l_i} C_{\kappa_{ik}}=\sum_{i=1}^{m}\sum_{k=1}^{l_i}\sum_{x=k}^{l_i} p_{i,\kappa_{ix}}=\sum_{i=1}^{m}\sum_{k=1}^{l_i} kp_{i,\kappa_{ik}}.$$

其中$p_{i,j}$为机器i上运行任务j需要的处理时间，实际上最后一个等号的意思是：倒数第k个任务对机器i上的所有任务完成时间和的贡献是其处理时间的k倍（很容易理解的）。

现在我们定义一个二分图$G=(V,E)$，其中$V=A\cup B$，A包含n个顶点，对应于n个任务v_j，B包含nm个顶点w_{ik}，代表机器i上倒数第k个处理的任务，显然$i=1...m,k=1...n$。边集E包含所有(v_j,w_{ik})，为每条边赋予权重：$(v_j,w_{ik})\to kp_{ij}$。下面证明在这个二分图上寻找到的最小完美匹配对应于最优调度。

证明：首先，每一个有效的调度策略都对应一个完美匹配（这是显然的），其次，并不是每个完美匹配都对应一个有效的调度策略，因为一个任务可能被指定为倒数第k个运行但是该机器上并没有满k个任务。但显然这样的匹配不是最小匹配，因为我们可以将k减小，以获得更小的权重。因此最小完美匹配一定对应一个有效的调度策略，且根据之前的分析，该匹配的权重和等于所有任务的完成时间总和，从而该策略是最优的。

关于这个问题的简化版$P||\sum C_j$，也可以通过上面的分析看出，每个机器上倒数第k个完成的任务j导致时间和增加了kp_j，因此基本思路就是长任务尽量晚地在机器上执行，也就是先将前m长的任务分配给各机器处理队列的尾部，重复上述步骤。这和最短任务优先SPT算法的结果一致，因此SPT算法不仅是$1||\sum C_j$的最优解，还是$P||\sum C_j$的最优解。

$O|pmtn|C_{max}$问题

第二个使用二分图匹配来设计算法的是$O|pmtn|C_{max}$问题，即开放车间下的抢占式调度优化最终完成时间。我们在之前使用LS算法解决过$O||C_{max}$问题，其中提到两个下界：Π_{max}是单个机器的最大负载时间（每个任务在单个机器上需求的处理时间之和），P_{max}是单个任务的最大处理时长。这两者仍然是抢占式调度下该问题的下界。实际上，允许抢占式调度的情况下，使用二分图匹配算法能够达到最优策略，其完成时间等于$\max(P_{max},\Pi_{max})$。

我们考虑对此问题的一个调度的任意时间点，在这个时间点，每个机器至多在处理一个任务，也就是说，在每个时间点，调度策略都指定了一个任务到机器的匹配。我们的目的是找到任意时间点的一个符合最优调度的匹配，并按照这个匹配来执行任务。

我们称总剩余处理时长等于P_{max}的任务为j_{max}，称机器负载时间达到Π_{max}的机器为m_{max}，也称它们是**紧迫的**，注意随着时间的推进，这两个值是变化的（有的任务可能不在匹配中，因为没执行）。如果能够让某时间点的匹配包含j_{max}和m_{max}，那么在这个匹配下执行t时间，我们就可以将这个调度的理论下界$\max(P_{max},\Pi_{max})$降低t。如果任意时间点的匹配均满足这一条件，那么最终能够将这个下界降低到零，这说明此时最大负载的机器已经处理完毕，所有任务都执行完毕，而处理时间恰好等于$\max(P_{max},\Pi_{max})$，这就是最优调度策略。同时这个匹配还必须满足所有边的剩余执行时间为正（这样才能继续执行），满足这些条件的匹配称为*decrementing set*，它的存在性证明较难，超出本文范畴，有兴趣的读者可以去读Lawler和Labetoulle的原论文[28]。

下面我们构造一个二分匹配问题来寻找它，这里比上一节的图简单。点集只需包含所有的任务i和机器j，同时包含每条边满足如下条件：连接一个机器i和一个任务j，且任务j在机器i上剩余所需的处理时间大于零。显然，根据上一段的分析我们需要这些边中包含进j_{max}和m_{max}，应用一些传统匹配算法很容易保证这一点。

现在我们找到了一个时间下的满足条件的匹配，只需按照这一匹配执行t时间，直到其中某个机器上的任务剩余时间等于零，或者某个机器剩余处理时间达到零，或者有新的任务或机器变紧迫（这是因为一个匹配可能不会包含所有任务和机器，导致其剩余处理时间没变），才寻找新的匹配。因此需要在许多时间点运行匹配算法，这会不会导致最终时间复杂度超过多项式时间呢？答案是不会。

证明：我们只会在特定条件（见上一段）满足时才寻找匹配，每个任务-机器对只会完成一次，因此最多需要运行nm次匹配，同时每个任务或机器只会变紧迫一次，因此最多$n+m$次。两者加起来不超过多项式时间。

为缩短文章长度，后续内容见调度算法（二）。

调度算法（二）

- 线性规划
  - $R|pmtn|C_{max}$问题

续调度算法（一）

线性规划

现在我们介绍线性规划算法在调度问题中的应用。一个线性规划问题通常以如下形式出现：

寻找长度为n的解向量$x=(x_1,...,x_n)$，满足m个线性约束$a_{i1}x1+a_{i2}x_2+...+a_{in}x_n \le b_i$，其中$1 \le i \le n$，并使得$c_1x_1+c_2x_2+...+c_nx_n$最小化，其中$c=(c_1,...c_n)$为成本向量。

有的时候，上面的一些约束可能是等式而非不等式，也有的时候甚至不存在需要最小化的目标函数，也就是说目标函数是任意的，我们只需要寻找到满足约束条件的解x即可。这些都算是线性规划问题的不同形式。许多优化问题都可以转化为线性规划求解，而线性规划问题存在多项式时间算法。本节主要考虑$R|pmtn|C_{max}$问题的线性规划模型。

$R|pmtn|C_{max}$问题

在不相关并行环境下进行抢占式调度，优化最终完成时间，不熟悉问题定义的建议回顾一下前面章节。抢占式调度下，单个任务可能在任何机器上运行一部分，为了形式化这个问题，我们使用nm个变量x_{ij}，代表任务j的总任务量在机器i上执行的比例。例如，如果$x_{1j}=x_{2j}=1/2$，则说明任务j在机器1和机器2上各完成了一半的任务量。

现在我们考虑怎样的线性约束能够使x_{ij}的解符合$R|pmtn|C_{max}$问题的定义。显然，每部分任务的比例必须是非负的，从而有：

同时，任意有效调度下，每个任务都必须被完成，这意味着：

我们的设计目标是最小化调度的最终完成时间D，显然任何机器上的总处理时间不超过D，根据之前的定义，不相关并行环境下，任务j在机器i上的处理时间记为p_{ij}：

最后，我们还需要保证，所有任务的总处理时间都不超过D：

我们需要在满足上述约束条件下最小化D。显然，任何一个有效调度下的x_{ij}必然满足这样的约束，但是有一事不明：一个满足约束条件的解并没有确切指定具体的调度策略——x_{ij}仅能指定任务在不同机器上运行的比例，而不能保证调度时同一时刻只能在一个机器上运行（这一条件无法写成线性约束）。

有趣的是，这个问题可以通过定义一个开放车间问题O来解决。我们为每个x_{ij}创建一个任务j在机器i上的操作，操作的处理时间为$o_{ij}=x_{ij}p_{ij}$，现在对于已经通过线性规划求出的最优的x_{ij}，我们构造了一个抢占式的开放车间问题，优化目标仍是C_{max}，此问题概括为$O|pmtn|C_{max}$，此问题已在上一节通过一个二分图匹配的形式化问题解决了，同时由于有D约束了任务的最长处理时间和机器的最长负载时间，也就是说P_{max},Π_{max}

$\leq D$，新问题的解就是原问题的最优调度。

线性规划问题不止这点应用，它还能用于设计NP难问题的近似算法，这将在下一节介绍。

使用松弛法设计近似算法

现在我们转向设计一些NP难的调度难题的近似算法。近似算法的设计通常基于相应的NP难问题的松弛版本。问题的**松弛**指的是从原问题中去除一些约束形成的新问题。例如$1|r_j, pmtn|\sum C_j$就是$1|r_j|\sum C_j$的松弛版本，因为实际上抢占式调度比非抢占式调度简单多了，更容易计算最优解。另一种松弛的思路是移除“一个机器只能同时处理一个任务”的条件，让一个机器可以在同一时间运行多个任务。

显然，原问题的解一定是松弛问题的解，反之不一定。我们在上面章节提到的所有非抢占式调度算法都能用于解决抢占式调度，其解虽然可能不最优，但合法。同样地，松弛问题的最优解即使是合法的，也不一定达到原问题的最优解。

在这一课题上一个有效的想法是：设计一个可以在多项式时间内解决的松弛问题，然后再设计一个将松弛问题的解转化为原问题可行解的算法，并保证解尽量接近最优解。问题的关键是设计的松弛问题尽可能包含原问题较多的信息，使得松弛问题的最优解与原问题最优解尽可能接近。

本节将介绍两种对调度问题的松弛方法，一是通过非抢占式到抢占式的松弛，二是通过构造线性规划问题，松弛某些线性约束来实现。将松弛解转化为原问题解的方法也有两种，一是根据任务与机器的配对，二是根据任务在机器上的执行顺序。后面会详细介绍。

在开始本节内容之前，我们先介绍一个概念：松弛决策过程（relaxed decision procedure, RDP）。一个最小化问题的ρ-RDP定义为，接受一个目标值T，如果松弛问题没有小于等于T的解则返回$None$，否则返回一个不超过ρT的原问题解。一个多项式时间的ρ-RDP能够很容易的转化为原问题的ρ-近似算法：通过对T进行二分查找，寻找最小的T，使得松弛问题有解并将此解转化为原问题的解。

为什么是ρ-近似的呢？逻辑是这样：假如原问题有最优解T^*，则松弛问题在T^*下必有解，将这个解转化为不超过ρT的原问题解，这个过程构成ρ-近似算法。下面将应用这个设计思路。

$R||C_{max}$问题

本节给出一个$R||C_{max}$的2-近似算法。我们刚刚用线性规划解决过的$R|pmtn|C_{max}$问题，是这个问题的抢占式松弛版本，实际上，如果我们加一条约束$x_{ij} \in \{0,1\}$，就是非抢占式调度的线性规划模型了。实际上加上这个条件后，原先的第3条约束可以使得第4条约束多余，因此同样的建模下，本问题的线性约束为：

$$\sum_{i-1}^{m} x_{ij} = 1,\ for\ j = 1, . . . , n \sum_{j=1}^{n} p_{ij} x_{ij} \leq D,\ for\ i = 1, . . . , m x_{ij} \in \{0,1\},\ for\ i = 1, . . . , m,\ j = 1, . . . , n$$

这是一个整数线性规划问题，相比于一般的线性规划问题，整数线性规划是NP难的，因此问题并没有变简单。但是，我们提出的模型是有用的，可以用来松弛某些约束，例如得到一个非整数解，然后舍入到整数。

现在我们再次将整数约束松弛为$x_{ij} \geq 0$，但这就太松了一点，我们还要添加一个约束：如果一个任务j在机器i上的效率过慢，即$p_{ij} \geq D$，就不把这个任务分配到机器i：

这个约束在原来的整数线性规划模型中是天然成立的，但我们移除整数约束后就需要

手动添加。注意到，当D固定时，这是个简单的只有线性约束的规划问题（没有成本函数），我们只要找可行解即可。如果对于给定的D，问题没有解，则输出$None$，如果有解，我们就尝试将非整数解转化为原问题的可行整数解。

根据一个线性规划问题的基本结论：我们可以找到一个此问题的可行解，其中最多只有$n+m$个正值。这$n+m$个正的x_{ij}指定了n个任务的去向，假设这些x_{ij}中等于1的数量为k，则k个任务已经被完全指定去向，剩余$n-k$个任务需要至少$2(n-k)$个x_{ij}指定去向（因为非整数意味着至少被分配在两台机器上），故$k+2n-2k \le m+n$，故$n-k \le m$，意味着至多有m个任务被分配在不止一台机器上。

现在，我们将所有$x_{ij}=1$的任务j直接分配到机器i，这部分调度记为S_1，剩余还有至多m个任务需要调度。我们简单地按照$x_{ij}>0$的解将任务j匹配到机器i，且保证每个机器至多一个任务（因为这部分任务不超过m个），这部分任务的调度记为S_2。关于S_2的存在性证明稍微有点复杂，感兴趣的请读[1]。

我们分析一下这样形成的调度策略的完成时间，它显然不超过S_1的完成时间加上S_2的完成时间。而由于x_{ij}是松弛问题的可行解，因此S_1的完成时间是$\le D$的，在S_2中，由于每个机器至多一个任务，且由于约束(4)，$x_{ij}>0$意味着$p_{ij} \le D$，从而S_2的完成时间也是不超过D的，故这样调度的时间不超过$2D$。因此假设原问题的最优解是D，则该松弛问题必有解，且该解不超过$2D$。

$1|r_j|\sum C_j$问题

前面已经提到，带发布时间的单机调度问题是NP难问题，本节将设计一个解决此问题的近似算法。对于此问题的抢占式版本，前面已经证明了最短剩余时间优先算法$SRPT$的最优性。利用这个松弛，我们将从其最优解中找出各任务的完成时间顺序，然后以相同的顺序构建非抢占式调度策略。此算法称为Convert-Preempt-Schedule，反转抢占式调度。

我们首先使用$SRPT$算法构建抢占式调度版本的最优解，然后按照该最优解中各任务的完成时间排序，按照$C_1^P \le ... \le C_n^P$给这些任务重新编号。而后按照相同顺序非抢占式的调度这些任务，如果某个时间点后继任务还未发布，就让机器空闲即可。可以证明，这个算法是2-近似的。

证明：对于任务j，由于其在抢占式调度中能在C_j^P时完成，因此假如没有其他任务，则它最晚也能在C_j^P时完成，当存在其他任务后，我们在其完成前插入了排在它前面的任务，因此它的最晚完成时间延长为：

而由于在抢占式调度中，所有$k \le j$的任务都在C_j^P之前完成了，因此$\sum_{k\le j} p_k \le C_j^P$，从而有：

假设非抢占式调度的最优解是$\sum C_j^P$，则对于抢占式的松弛版本必然有不超过$\sum C_j^P$的最优解，而通过上述分析可知此算法给出的解不超过抢占式最优解的两倍，证毕。

$1|r_j, prec|\sum w_j C_j$问题

这一节依然使用松弛法来解决这个难题（基本上是目前遇到的最难的单机调度问题了），我们采用线性规划来设计一个2-近似算法。

首先我们描述这个问题的线性规划约束，和之前不同，我们不从整数/非整数的角度松弛，而是从问题定义中寻找必要条件，形成一个线性规划问题。

对这个问题来说，任务运行的顺序极其重要，因此我们要寻找一种能表示任务顺序的形式化描述。很容易想到用时间来表示：定义C_j为任务j在调度策略中的完成时间。实际上这就能够完确定一个调度了（比之前的x_{ij}更清晰）。显然，最小化的成本函数是

$\sum w_jC_j$。并且满足如下条件：

$$C_j \geq r_j + p_j,\ j = 1, ..., n. C_k \geq C_j + p_k,\ for\ each\ j < k. C_k \geq C_j + p_k\ or\ C_j \geq C_k + p_j,\ for\ each\ j \neq k.$$

这三个条件很容易理解，首先任务必须在发布时间后才能开始，其次需要等待任务所有的前驱任务完成才能开始，最后，任意两个不同的任务存在先后关系（处理时间不重叠）。

不幸的是，最后一个约束有"或"运算，不是一个确定的线性约束，这导致这个问题不是线性规划问题。我们尝试用不等式代替它。定义任务集$J = \{1, ..., n\}$，对任意子集$S \subseteq J$，定义$p(S) = \sum_{j\in S} p_j$，$p^2(S) = \sum_{j\in S} p_j^2$。对任何可行的单机调度策略，可以证明下面的不等式成立：

$$\sum_{j\in S} p_jC_j \geq \frac{1}{2}(p^2(S) + p(S)^2),\ for\ each\ S \subseteq J.$$

证明：假设S中的任务按照完成时间升序编号。由于任意的任务j的完成时间大于在其前面完成的所有任务的处理时间之和，即$C_j \geq \sum_{k=1}^{j} p_k$，代入上式左边得：

$$\sum_{j\in S} p_jC_j \geq \sum_{j\in S} p_j \sum_{k=1}^{j} p_k = \frac{1}{2}(p^2(S) + p(S)^2)$$

对于没有发布时间，没有任务依赖的问题$1||\sum w_jC_j$问题，上面这个不等式的约束是完备的[37,41]，可以形成一个可行解。现在把上面这个不等式加上原先的第一个和第二个不等式，组成$1|r_j, prec|\sum w_jC_j$的线性约束，这个新的线性规划问题是原问题的松弛问题。

注意到我们新提出来的不等式似乎有2^n个，但是它仍然可以被线性规划的椭圆算法[37,41]在多项式时间内解决。

现在为了简单起见，我们移除发布时间约束，只探讨$1|prec|\sum w_jC_j$问题，其对应的松弛问题将移除上面提到的第一个不等式。解上面的线性规划问题得到的解为$C_1 \leq C_2 \leq ... \leq C_n$，我们将证明按照这个顺序来调度任务是2-近似的。

假设上面的调度形成的新的完成时间为$\widetilde{C_j}$，按照惯例，我们只需证明$\widetilde{C_j} \leq 2C_j$就行。首先，设$S = \{1, ..., j\}$，没有了发布时间机器是没有空闲的，因此$\widetilde{C_j} = p(S)$。同时根据上面提到的不等式$(*)$，我们有：

$$\sum_{k=1}^{j} p_jC_j \geq \frac{1}{2}(p^2(S) + p(S)^2) \geq \frac{1}{2}p(S)^2$$

又因为$C_1 \leq C_2 \leq ... \leq C_n$，我们有：

$$C_jp(S) = C_j \sum_{k=1}^{j} p_k \geq \sum_{k=1}^{j} C_kp_k \geq \frac{1}{2}p(S)^2$$

故$\widetilde{C_j} = p(S) \leq 2C_j$，证毕。