

## ジョブ制御プログラム

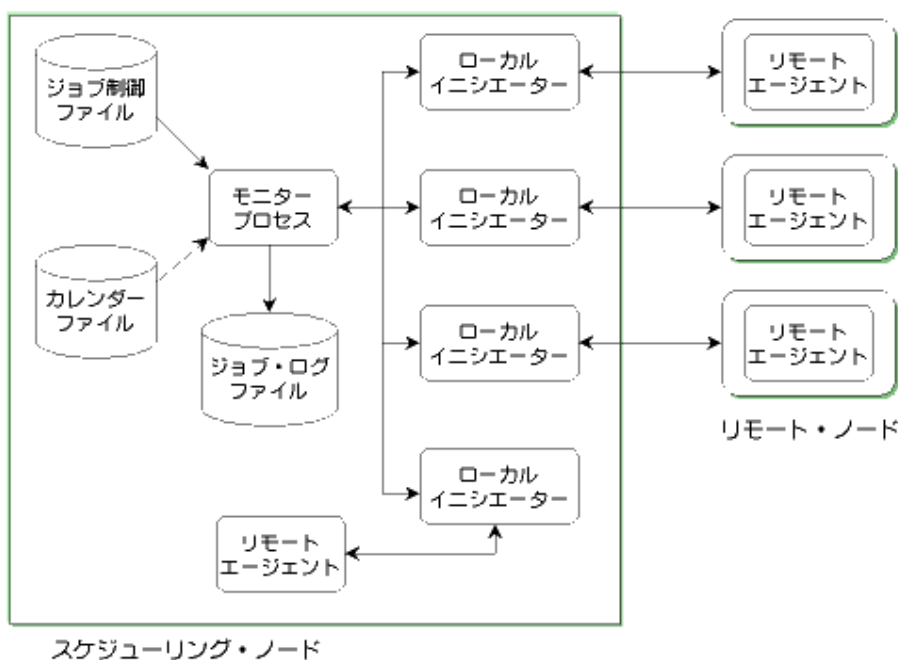
ジョブ制御プログラムは、メインフレームと同様のジョブ・ネットワーク定義によるジョブ・スケジューリングを実現します。

### 【 主な機能 】

- 前ジョブの実行結果により後続ジョブの実行を制御します。
- 指定時刻(日時)にジョブを実行します。
- 指定時間を越えて実行しているジョブをキャンセルします。
- ジョブが異常終了した場合には、システム・コンソール通知します。
- ジョブの実行結果をログに記録します。
- ウォーム・スタート/コールド・スタートをサポートします。
- ジョブ・クラスをサポートします。
- リモート・ホストでのジョブの実行をサポートします。

### 【 ジョブ制御プログラムの概要 】

ジョブ制御プログラムは、モニター・プロセス、ローカル・イニシエーター、リモート・エージェントの3種類のプロセスで稼働します。



モニター・プロセスは、ジョブ制御プログラムの中心となるプロセスで、ジョブ制御ファイルを読みこんでジョブ・ネットワークに基づいたスケジューリングを行ないます。ジョブの実行状況は、モニター・プロセスを通じてジョブ・ログファイルに書き込まれます。

モニター・プロセスはスタートするとジョブ制御ファイルの指定に従ってローカル・イニシエーターを複数スタートさせます。各ローカル・イニシエーターはリモート・ノード定義に従って、リモート・ノードのリモート・エージェントと通信を開始。スケジュールさ

れたジョブにより実際に実行されるプログラムは、ローカル・イニシエーター経由でリモート・エージェントに指示され、リモート・ノード上で実行されます。

### ジョブの依存関係

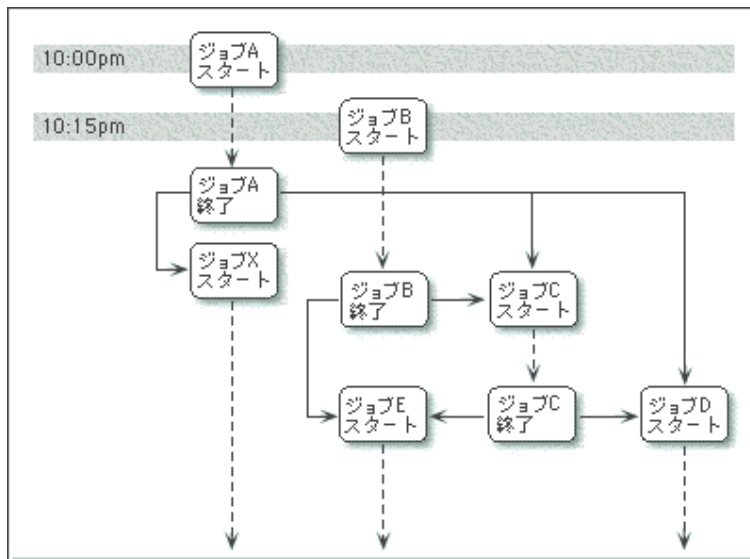
ジョブ制御プログラムに定義される各ジョブは単独での定義も可能ですが、多くの場合はジョブ・ネットワークに従った依存関係を持ちます。

### ジョブの実行クラス

各ジョブは1つの実行クラスが定義されます。各イニシエーターは、最大4つの実行クラスが定義され、それぞれに定義された実行クラスと一致するクラスのジョブを順次実行します。このことにより、複数のイニシエーターに定義されたクラスに属するジョブは実行される確立が高まり、結果的には他のジョブより優先処理されます。

## 【 分散環境下でのバッチ処理 - スケジューリング 】

ジョブ制御プログラムでバッチ処理を行なう場合の例



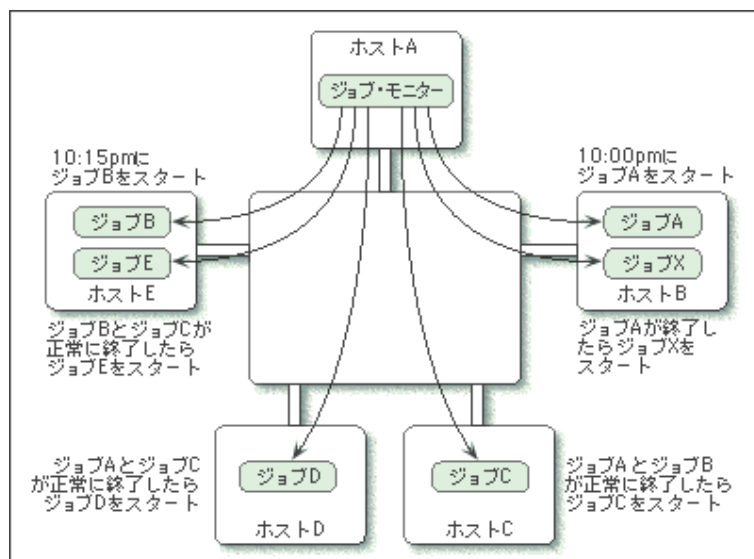
1. バッチ処理のスケジュールのサンプルを考えます。
2. 時刻指定で動くジョブが2つあります。  
ジョブ A は、10:00pm にスタート。もう1つの時刻指定ジョブであるジョブ B は、10:15pm にスタート。後続ジョブが前提ジョブの終了に従って実行されます。
3. ジョブ X は、ジョブ A の終了後直ちに実行開始。
4. ジョブ C はジョブ A とジョブ B の終了を待って実行されます。
5. ジョブ D はジョブ A とジョブ C の終了後に、ジョブ E はジョブ B とジョブ C の両方が終了後に実行されるものとして。

このようなスケジューリングをシェル・スクリプトだけで組むのは容易ではありません。ネットワーク上の複数の AIX で分散させて実行する事はさらに困難です。例えば、ジョブ A とジョブ B がともに CPU バウンダリーのジョブだった場合、同じマシン上で実行すると

お互いに CPU を取り合い、その結果エラー・タイムが単独で実行した時よりも延びてしまいます。これを別々の AIX 上で実行させれば、本来の処理時間でそれぞれのジョブを終了させることができます。

## 【 分散環境下でのバッチ処理 - 実行 】

ジョブ・スケジュールを実際に分散環境下で実行した場合

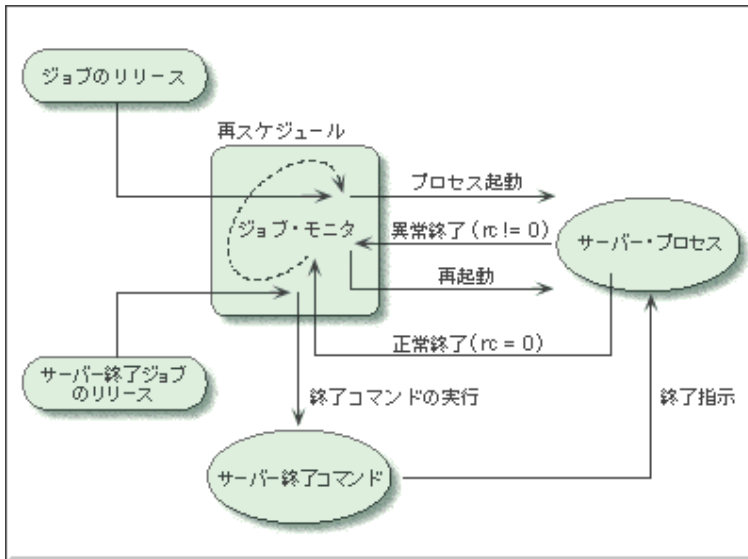


ジョブ・モニターはホストAでジョブのスケジューリングを行ないます。

1. 10:00pm にジョブ A をホスト B で実行。
2. 10:15pm にジョブ B をホスト E で実行。
3. ジョブ A が終了すると後続ジョブで実行条件のそろっているジョブ X が直ちに実行されます。上記の場合は、ジョブ A と同じくホスト B で実行されています。
4. ジョブ B が終了すると、ジョブ A とジョブ B の終了を待っていたジョブ C はジョブ X と平行処理される目的でホスト C にスケジュールされ、ジョブ C の終了を待ってジョブ D がホスト D で、ジョブ E がホスト E でそれぞれ実行されます。これらのジョブもジョブ X と独立して平行処理できるので、それぞれ異なるホストにスケジュールします。

### 【 サーバー・プロセスの監視 】

システム稼働中は常駐すべきサーバー・プロセスを、繰返しスケジュールされるジョブとして定義することでプロセスの監視・再起動が可能です。



リピート定義（再スケジュールリングの定義）されたジョブは、ジョブ・モニターの起動時にホールド・ジョブとしてスケジュールされます。ホールド状態を解きサーバー・プロセスを起動するには、`jobcmd` コマンドでジョブをリリースします。途中、プロセスが異常終了 (`rc != 0`) した場合は、リターン値が 0 意外の時の後続ジョブとして、再度サーバーのジョブをリリースする `jobcmd` を定義することで自動的なサーバー・プロセスの再起動を行なう事ができます。

また、直接サーバー終了コマンドをオペレーターが実行しなくても、サーバー終了ジョブとして定義する事で該当のジョブをリリースすればサーバーを正常に停止することができます。

## 【 定義ファイル例 】

ジョブ制御プログラムには定義ファイルにてジョブの起動日時や起動条件、リモート・ノードの設定、休日の指定を定義することができます。

### ジョブ制御ファイル

ジョブ制御ファイルは、時刻や実行コマンド、クラスなどが「項目 = 値」で分かりやすく記述できます。

```
#      JOB Shedule Table
#
maxprocs      = 4  # MAX No. of tasks to run concurrently
maxjobs       = 10 # MAX No. of jobs to be scheduled
textarea      = 3  # "program", "message" text area
maxdependency = 8  # MAX No. of dependencies
init_class    = AB,FY,C,AZ # class
jobxxx:
daily         = 14:35
program       = '/home/toshiaki/jobctl/dev4/xxx'
class         = A
succesxxx:
jobxxx        = 0
program       = 'ksh -c "echo `date` && banner jobxxx ok"'
class         = F
failxxx:
jobxxx        > 0
program       = 'ksh -c "echo `date` && banner jobxxx ng"'
class         = Y
tar:
succesxxx:    = 0
weekly        = friday
clock         = 22:00
program       = '/usr/bin/tar cvf /dev/rmt0 /usr/lib/X11/fonts'
time          = 300
class         = C
alart         = console
message       = 'JOB Name [tar] abend...'
succestar:
tar           = 0
program       = 'ksh -c "echo `date` && banner tar success"'
class         = Z
failtar:
tar           > 0
program       = 'ksh -c "echo `date` && banner tar failed"'
class         = Z
```

### リモート・ノード・ファイル/カレンダー・ファイル

リモート・ノード・ファイルはジョブを実行したいリモート・ノードのホスト名を記述します。上から順番にイニシエーター番号が割り当てられます。

カレンダー・ファイルは、休日制御を行ないたい場合に使用します。休日は、曜日、月日、月と曜日等の形式で指定できます。

### リモート・ノード・ファイル

```
oscar  
whale  
ois1  
inoki
```

### カレンダー・ファイル

```
saturday  
sunday  
1/1  
1/2  
1/3  
1/monday(2)  
2/11  
4/29  
5/3  
5/4  
...
```