

# Xcrypt Manual

E-Science Group, Nakashima Laboratory, Kyoto University

January 20, 2010

# Contents

<b>I</b>	<b>General</b>	<b>3</b>
<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Overview . . . . .	4
1.2	Environment . . . . .	4
<b>2</b>	<b>Script</b>	<b>5</b>
2.1	Module . . . . .	5
2.2	Template . . . . .	6
2.3	Procedure . . . . .	6
2.4	Example . . . . .	6
<b>3</b>	<b>Flow</b>	<b>8</b>
3.1	Model . . . . .	8
3.2	Execution . . . . .	8
3.3	Product . . . . .	9
<b>II</b>	<b>Details</b>	<b>11</b>
<b>4</b>	<b>Module</b>	<b>12</b>
4.1	core . . . . .	12
4.2	limit . . . . .	12
4.3	successor . . . . .	13
4.4	n_section_method . . . . .	13
4.5	dry . . . . .	13
<b>5</b>	<b>Template</b>	<b>15</b>
5.1	id . . . . .	15
5.2	exe . . . . .	15
5.3	arg <i>i</i> . . . . .	15
5.4	linkedfile <i>i</i> . . . . .	15
5.5	copiedfile <i>i</i> . . . . .	15
5.6	copieddir <i>i</i> . . . . .	16
5.7	stdofile . . . . .	16
5.8	stdefile . . . . .	16
5.9	queue . . . . .	16
5.10	cpu . . . . .	16
5.11	proc . . . . .	16

5.12	option . . . . .	16
<b>6</b>	<b>Built-in Function</b>	<b>17</b>
6.1	prepare . . . . .	17
6.2	submit . . . . .	19
6.3	sync . . . . .	20
6.4	addkeys . . . . .	21
6.5	addperiodic . . . . .	21
6.6	prepare_submit . . . . .	21
6.7	submit_sync . . . . .	21
6.8	prepare_submit_sync . . . . .	21
<b>A</b>	<b>ジョブクラス拡張モジュール実装の手引</b>	<b>22</b>
A.1	はじめに . . . . .	22
A.2	拡張モジュールの定義と利用 . . . . .	22
A.3	拡張モジュールスクリプトの構成 . . . . .	22
A.4	特別な意味を持つメソッド . . . . .	24
A.4.1	new メソッド . . . . .	24
A.4.2	before_isready メソッド . . . . .	24
A.4.3	before メソッド . . . . .	25
A.4.4	start メソッド . . . . .	25
A.4.5	after_isready メソッド . . . . .	25
A.4.6	after メソッド . . . . .	25
A.4.7	before_isready , before , start , after_isready , after の並行性 . . .	25

## Memo Xcrypt requires

- Marc Lehmann’s AnyEvent, common::sense, Coro, EV, and Guard,
- Joshua Nathaniel Pritikin’s Event, and
- Daniel Muey’s Recursive.

**Part I**

**General**

# Chapter 1

## Introduction

### 1.1 Overview

In using a high-performance computer, we usually commit job-processing to a job scheduler. At this time, we often go through the following procedure:

- to create a script in its writing style depending on what the job scheduler is,
- to pass the script to the job scheduler, and
- to extract data from its result, to create another script from the data, and to pass it to the job scheduler.

However, such a procedure requires manual intervention cost. It therefore seems better to remove manual intervention in mid-processing by using an appropriate script language. Xcrypt is a script language for job parallelization. We can deal with jobs as objects (called *job objects*) in Xcrypt and manipulate the jobs as well as objects in an object-oriented language. Xcrypt provides some functions and modules for facilitating job generation, submission, synchronization, etc. Xcrypt makes it easy to write scripts to process job, and supports users to process jobs easily.

### 1.2 Environment

Xcrypt requires a superset of Bourne or C shell, Perl 5.10.0 or any later version, and Perl/Tk 8.4 for GUI.

## Chapter 2

# Script

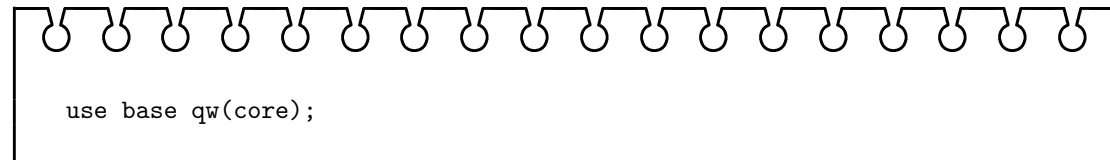
Xcrypt is a script language, an extension of Perl. Xcrypt provides some functions and modules (not in Perl) which supports how to deal with *jobs*.

An Xcrypt script consists of descriptions of

1. Module,
2. Template, and
3. Procedure.

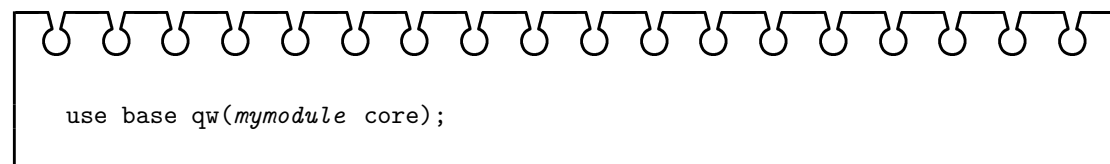
### 2.1 Module

Modules for job objects are used as follows,



```
use base qw(core);
```

When you use multiple modules, it is enough to write



```
use base qw(my module core);
```

Every module is used in order. In Chapter 4, modules are explained in detail.

Ordinal modules are used as follows,

```
use builtin;
```

similarly to how to use modules in Perl.

Some modules define variables. In order to use such variables, it is enough to write e.g.,

```
$separator = '-';
```

## 2.2 Template

Xcrypt's templates are implemented as Perl's hashes. For example,

```
%myjob = (  
  'id' => 'myjob',  
  'exe' => './myexe',  
  'arg0' => '100',  
  'arg1' => 'myinput',  
  'linkedfile0' => 'myexe',  
  'copiedfile0' => 'myinput',  
  'stdofile' => 'myout',  
  'stdefile' => 'myerr',  
  'queue' => 'myqueue',  
  'option' => 'myoption'  
);
```

Keys in templates are described in Chapter 5 in detail.

## 2.3 Procedure

Procedure (usually manually done) are written in Xcrypt (and Perl). Xcrypt's functions are described in Chapter 6.

## 2.4 Example

An example script is as follows,

```
use base qw(limit core);

&limit::initialize(10);
$separator = '-';

%myjob = (
    'id' => 'myjob',
    'exe' => './myexe',
    'arg0' => '100',
    'arg1' => 'myinput',
    'linkedfile0' => 'myexe',
    'copiedfile0' => 'myinput',
    'stdofile' => 'myout',
    'stdefile' => 'myerr',
    'queue' => 'myqueue',
    'option' => 'myoption'
);
&prepare_submit_sync(%myjob, 'arg0@' => [2,4]);
```



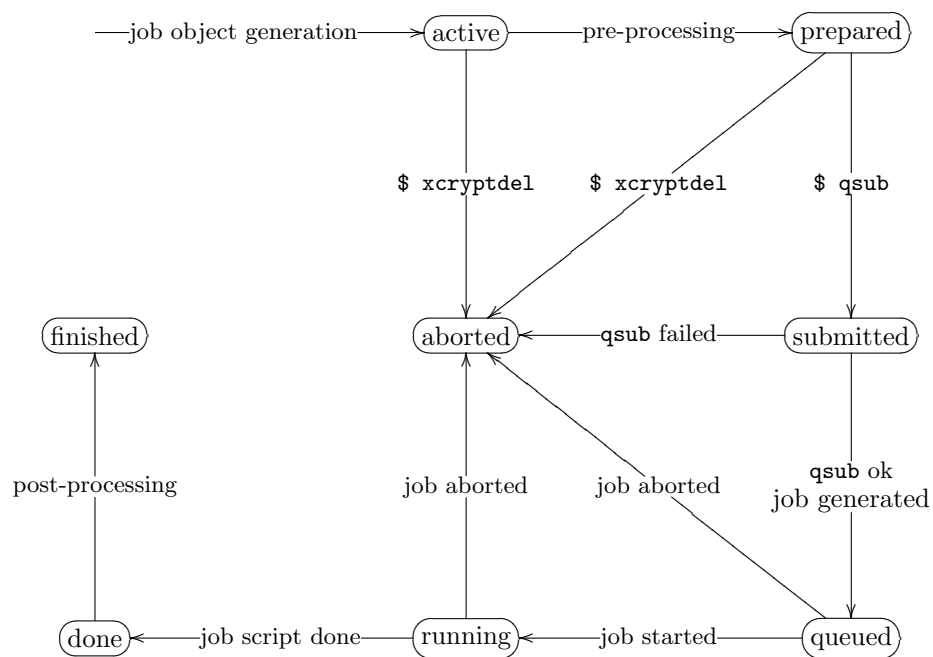
# Chapter 3

## Flow

In this chapter, we introduce how jobs are processed.

### 3.1 Model

Any job transits to *states* as follows,



### 3.2 Execution

Set the following environment variables appropriately. **XCRYPT** is set to the directory (e.g., `/usr/share/xcrypt`) in which Xcrypt is installed. For example, it is enough to type

```
$ XCRYPT=/usr/share/xcrypt; export XCRYPT
```

when you use Bourne shell. `XCRJOBSCHED`<sup>1</sup> is set to a job scheduler. `PERL5LIB` is set to `$XCRYPT/lib` and `$XCRYPT/lib/algo/lib`.

Next, move to the working directory (e.g., `$HOME/wd`)

```
$ cd $HOME/wd
```

and write an Xcrypt script (e.g., `sample.xcr`). See Section 2.4 in order to know how to write.

Finally, execute Xcrypt with the script:

```
$ $XCRYPT/bin/xcrypt sample.xcr
```

### 3.3 Product

Xcrypt creates the following in the working directory just/after then.

#### directory

is created each job (called a *job working directory*). It is named by the value of the job hash's key `id`. Job-processing is done in the job working directory.

#### *myscript.pl*

is a Perl script created from the Xcrypt script *myscript*.

#### file and soft link

of the names (which are the values of `copiedfilei` and `linkedfilei`) are copied and created in the job working directory, respectively.

#### `$XCRJOBSCHED.sh`

is a job script, which is passed to a job scheduler.

#### `sh.sh`

is a Bourne shell script, which is executed, regarding OS as a job scheduler.

#### `request_id`

is a file storing the job's request ID (which the job scheduler returns).

#### `stdout`

is a file storing the job's standard output. When `stdofile` is defined, the file is renamed as its value.

---

<sup>1</sup>`NQS`, `hNQS`, `SGE`, and `tSGE` are available. In addition, sh-mode is also available (by being set to `sh`) for environments without any job scheduler. In that case, jobs are dealt with as processes in OS.

**stderr**

is a file storing the job's standard error. When **stdefile** is defined, the file is renamed as its value.

# Part II

## Details

# Chapter 4

## Module

In this chapter, we introduce some modules available in Xcrypt scripts.

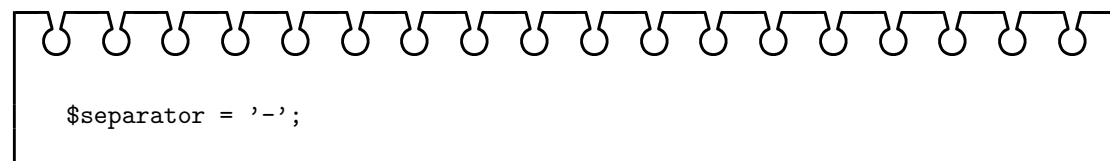
### 4.1 core

This module is the Xcrypt core module, and required to be read in order to use anything particular to Xcrypt.

It creates a directory of the name

```
$myjob->{'id'} $separator $myjob->{'arg0'} $separator ...
```

under the job working directory, where %myjob is a template. The default of \$separator is .. In order to redefine \$separator to be -, for example, it is enough to write as follows,



```
$separator = '-';
```

Any word of ASCII printable characters except

```
@_["$%&'/:;<=>?[\]`{|}
```

is available.

This module also makes the variable \$separator\_nocheck to be available in Xcrypt scripts. When the value of \$separator\_nocheck is 1, Xcrypt skips to check \$separator for availability. The default is 0.

### 4.2 limit

This module limits the number of jobs to be submitted. In order to limit the number of jobs to 10, for example, it is enough to write as follows,

```
&limit::initialize(10);
```

### 4.3 successor

This module provides how to write dependency of jobs declaratively. For example, in order to define job objects of the name `%x`, `%y`, write:

```
...  
'successors' => ['x', 'y'],  
...
```

using the key `successors` in the template.

### 4.4 n\_section\_method

This module provides *n*-section method, a root-finding algorithm of the only difference from bisection method<sup>1</sup> is the number of sections.

The values `partition` and `epsilon`, denote a partition number and an error, respectively. An interval is expressed by `x_left` and `x_right`. The values `y_left` and `y_right` are values on `x_left` and `x_right`. Typically, we can call the function `n_section_method` with these keys, e.g.,

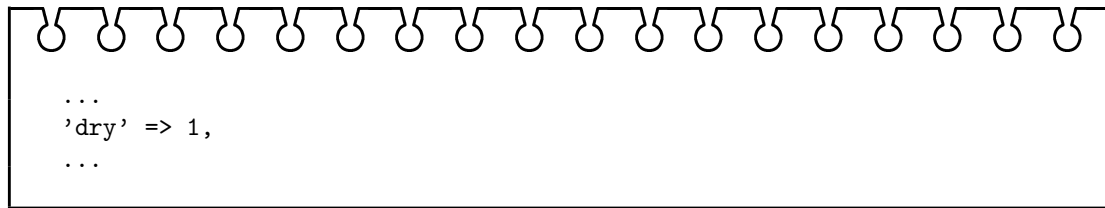
```
&n_section_method::n_section_method(%job,  
  'partition' => 12, 'epsilon' => 0.01,  
  'x_left'    => -1,  'x_right' => 10,  
  'y_left'    => 0.5, 'y_right' => -5  
);
```

### 4.5 dry

This module provides job-processing in dry mode (skipping any command execution). Description in a template

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Bisection\\_method/](http://en.wikipedia.org/wiki/Bisection_method/)



```
...  
'dry' => 1,  
...
```

makes any job (derived from this hash) to process in dry mode.

# Chapter 5

## Template

In this chapter, we introduce keys and values available in templates by default.

### 5.1 id

Its value is a word. The value is used for creating job objects and identifying the job objects as their prefixes. Any word of ASCII printable characters except

@\_ "\$%&' / : ; < = > ? [ \ ] ' { | }

is available.

### 5.2 exe

Its value denotes a command. The command is executed as follows,

```
$ myexe myarg0 myarg1 ... myarg255
```

with `argi` as explained below.

### 5.3 arg<sub>*i*</sub>

Its values are arguments of a command. The command is executed as follows,

```
$ myexe myarg0 myarg1 ... myarg255
```

### 5.4 linkedfile<sub>*i*</sub>

A soft link of the file (whose name is its value) is created in the job working directory.

### 5.5 copiedfile<sub>*i*</sub>

A file of the name its value is copied in the job working directory.



## **5.6   `copieddir`**

All files in the directory of the name its value are copied in the job working directory.

## **5.7   `stdofile`**

The standard output is stored in a file of the name its value. The default is `stdout`.

## **5.8   `stdefile`**

The standard error is stored in a file of the name its value. The default is `stderr`.

## **5.9   `queue`**

Its value is a name of a queue.

## **5.10   `cpu`**

Its value is the number of cores used exclusively.

## **5.11   `proc`**

Its value is the number of processes used exclusively.

## **5.12   `option`**

Its value is an option of a job scheduler.

## Chapter 6

# Built-in Function

In this chapter, we introduce built-in functions.

### 6.1 prepare

This function takes a job definition hash and parameters of references<sup>1</sup>, and returns an array of job objects. The ids of job objects are generated by `RANGEi` as described later.

#### Format

```
prepare(%template
    [, 'RANGE0' => \@myparam0s]...[, 'RANGE255' => \@myparam255s]
    [, 'key@' => \@myparams]...);
```

where `key@` denotes the one whose postfix is the character `@` (e.g., `arg0@`).

Any word of ASCII printable characters except

`@_"$%&'/:;<=>?[\\'{}|}`

is available for `RANGEi`'s values.

#### Example

```
@jobs = prepare('id' => 'myjob', 'exe' => './myexe', 'arg0@' => [10,20]);
```

---

<sup>1</sup>In this manual, references do not denote type globs.

This is almost the same as

```
@jobs = ();  
push(@jobs, {'id' => 'myjob_0', 'exe' => './myexe', 'arg0' => '10'});  
push(@jobs, {'id' => 'myjob_1', 'exe' => './myexe', 'arg0' => '20'});
```

Declarative description is as follows,

```
%template = ('id' => 'myjob', 'exe' => './myexe', 'arg0@' => [10,20]);  
prepare(%template);
```

Description of a template can be separated from parameters:

```
%template = ('id' => 'myjob', 'exe' => './myexe');  
prepare(%template, 'arg0@' => [10,20]);
```

Various job objects can be generated by using `RANGE`'s. For example,

```
@jobs = prepare(%template, 'RANGE0' => [0..99], 'arg0@' => '2 * $R0');
```

is the same as a sequence of `prepare(%template, 'arg0' => 0)`, `prepare(%template, 'arg0' => 2),...`, `prepare(%template, 'arg0' => 198)`.

### Advanced

It is possible to generate job objects by using multiple parameters. For example,

```
%template = ('id' => 'myjob', 'exe' => './myexe');  
@jobs = prepare(%template, 'arg0@' => [0,1], 'arg1@' => [2,3]);
```

is almost the same as

```
%template = ('id' => 'myjob', 'exe' => './myexe');
@jobs = ();
@job0 = prepare(%template, 'arg0' => '0', 'arg1' => '2');
push(@jobs, $job0[0]);
@job1 = prepare(%template, 'arg0' => '1', 'arg1' => '3');
push(@jobs, $job1[0]);
```

To use `RANGE`s makes it to generate job products by multiple parameters, e.g.,

```
%template = ('id' => 'myjob', 'exe' => './myexe');
@jobs = prepare(%template, 'RANGE0' => [0,1], 'RANGE1' => [2,4],
                  'arg0@' => '$R0 + $R1');
```

is almost the same as

```
%template = ('id' => 'myjob', 'exe' => './myexe');
@jobs = ();
@job0 = prepare(%template, 'arg0' => '0', 'arg1' => '2');
push(@jobs, $job0[0]);
@job1 = prepare(%template, 'arg0' => '0', 'arg1' => '4');
push(@jobs, $job1[0]);
@job2 = prepare(%template, 'arg0' => '1', 'arg1' => '2');
push(@jobs, $job2[0]);
@job3 = prepare(%template, 'arg0' => '1', 'arg1' => '4');
push(@jobs, $job3[0]);
```

## 6.2 submit

This function takes an array of job objects, passes the jobs (corresponding to the job objects) to a job scheduler. Its return value is no change (the array of job objects).

### Format

```
submit(@myjobs);
```

### Example

Typically, this function takes a return value of `prepare`.

```
@jobs = prepare('id' => 'myjob', 'exe' => './myexe', 'arg0' => [10,20]);  
submit(@jobs);
```

It is possible to define job references without using `prepare` (though not to be recommended).

```
submit('id' => 'myjob_0', 'exe' => './myexe', 'arg0' => '10',  
      'id' => 'myjob_1', 'exe' => './myexe', 'arg0' => '20');
```

## 6.3 sync

This function takes an array of job objects, syncs the job objects. Its return value is no change (the array of job objects).

### Format

```
sync(@myjobs);
```

### Example

Typically, this function takes `prepare` (the same as `submit`).

```
@jobs = prepare('id' => 'myjob', 'exe' => './myexe', 'arg0@' => [10,20]);  
submit(@jobs);  
sync(@jobs);
```

## 6.4 addkeys

This function takes an array of words and makes it available as keys in job definition hashes.

### Format

```
addkeys(@words);
```

## 6.5 addperiodic

This function takes a Xcrypt's script code (written as *mystring*) and an integer *i*, and evaluates it each the *i* seconds.

### Format

```
addperiodic(mystring, i);
```

## 6.6 prepare\_submit

This function is an abbreviation of `prepare` and `submit`. Its format follows `prepare`.

## 6.7 submit\_sync

This function is an abbreviation of `submit` and `sync`. Its format follows `submit`.

## 6.8 prepare\_submit\_sync

This function is an abbreviation of `prepare_submit` and `sync`. Its format follows `prepare`.

## Appendix A

# ジョブクラス拡張モジュール実装の手引

### A.1 はじめに

Xcrypt の `prepare` 関数によって生成された全てのジョブオブジェクトは `$XCRYPT/lib/core.pm` で定義された `core` クラスに属する。Xcrypt のユーザや開発者は、拡張モジュールを定義して `core` クラスを拡張することで Xcrypt の機能を拡張することが可能である。本章では、この拡張モジュールを開発するために必要な情報を示す。

### A.2 拡張モジュールの定義と利用

*mymodule* という名前の拡張モジュールを定義するためには、*mymodule.pm* という名前のファイルを `$XCRYPT/lib/` (または `$PERL5LIB` で指定されているどこかのディレクトリ) に置く。Xcrypt ユーザは、スクリプトの先頭で、

```
use base (... mymodule ... core);
```

のようにモジュール名を指定することで、当該モジュールの機能を使用できる。

### A.3 拡張モジュールスクリプトの構成

典型的な拡張モジュールの定義スクリプトは以下のように記述される。

```

package mymodule;

use strict;
use ...;

&addkeys('my_instance_member', ...);

my $my_class_member;

# special methods
sub new {
    my $class = shift;
    my $self = $class->NEXT::new(@_);
    ...
    return bless $self, $class;
}

sub before_isready { ... }
sub before { ... }

sub start
{
    my $self = shift;
    ...
    $self->NEXT::start();
    ...
}

sub after { ... }
sub after_isready { ... }

# general methods
sub another_method
{
    ...
}

```

以下、スクリプトの各構成要素について説明する。

1. モジュール名の定義: `package` で指定する。ここで指定する名前は、モジュールファイル名の拡張子 ( `.pm` ) を除いた部分と同一でなければならない。
2. Perl モジュールの取り込み: 通常の Perl プログラムと同様に、必要な Perl モジュールを `use` で取り込む。
3. 追加するインスタンス変数の定義: ジョブクラスに新たに追加したいインスタンス変数の名前を `addkeys` 関数により定義する。ここで定義したインスタンス変数は、ジョブオブジェクトの属



性として、Xcrypt スクリプトやモジュール内のメソッドから `$job->{my_instance_member}` などとしてアクセスできる。また、Xcrypt ユーザがジョブ定義ハッシュの記述時に、  

```
%template = { ..., my_instance_member=>value, ... }
```

 のようにして値を設定することもできる。

4. クラス変数の定義：通常の Perl のオブジェクト指向プログラミングと同様、クラス変数はパッケージ内のグローバル変数として定義する。ここで定義した変数は、`$mymodule::my_class_member` としてアクセスできる
5. メソッドの定義：このモジュールにおいて追加、拡張するメソッドをパッケージ内のトップレベル関数として定義する。通常の Perl オブジェクト指向プログラミングと同様であるが、特定の名前を持つメソッドは特別な意味を持つので注意する（次節で説明）

## A.4 特別な意味を持つメソッド

### A.4.1 new メソッド

コンストラクタに相当するクラスメソッドであり、Xcrypt の `prepare` 関数（6.1 節）の処理中に、最も specialized なクラス（Xcrypt スクリプトのヘッダで宣言されているモジュール列のうち最も左に書かれているもの）の `new` メソッドが呼び出される。

この `new` メソッドに渡される引数は以下の通りである。

1. Xcrypt スクリプトが属するパッケージ名（= `user`）
2. ジョブオブジェクトへの参照。オブジェクトのメンバには、`prepare` 関数に渡されたジョブ定義ハッシュに対応する値がセットされている。

ジョブ定義ハッシュの `RANGE` 等の指定により `prepare` 関数が複数のジョブオブジェクトを生成した場合は、その各オブジェクトに対して `new` が適用される。

メソッドの本体では、`$class->NEXT::new($self,$obj)`（`$class`、`$obj` はそれぞれ引数として渡されたクラス名およびオブジェクトへの参照）のようにして親クラスの `new` メソッドを呼び出すことができる。典型的には、各 `new` メソッドはまず親クラスの `new` メソッドを、与えられた 2 つの引数をそのまま引き渡して呼び出し、その戻り値のオブジェクトにアクセスしつつ必要な処理を行った後、`bless` オブジェクトへの参照、渡されたクラス名の値をメソッドの戻り値として `return` すべきである。

`core` モジュールでも `new` メソッドが定義されており、ここでは 3.3 節で説明したジョブの作業ディレクトリおよびその中へのファイルのコピー・シンボリックリンクの生成処理を行う。子クラスの `new` メソッドにより `core` モジュールの `new` メソッドが呼び出されないと、この処理が行われなくなるので注意すること。

### A.4.2 before\_isready メソッド

状態（3.1 節）が `prepared` になったジョブオブジェクトに対して適用される。一般には Xcrypt スクリプト中の `submit` 関数（6.2 節）の適用により、ジョブの状態が `prepared` になる。引数はジョブオブジェクトへの参照である。複数のモジュールで `before_isready` メソッドが定義されていた場合は、子クラス 親クラスの順で全ての `before_isready` メソッドが呼び出される。

各メソッドは、真偽値を返す。呼び出された `before_isready` メソッド列のうち、1 つでも偽を返したメソッドがあれば、しばらくの時間待ち合わせた後、もう一度 `before_isready` メソッド列の実行が行われる。

### A.4.3 before メソッド

before\_isready メソッド列が適用され、その全てが真を返したジョブオブジェクトに対して適用される。引数はジョブオブジェクトへの参照である。複数のモジュールで before メソッドが定義されていた場合は、子クラス 親クラスの順で全ての before メソッドが呼び出される。

メソッドの戻り値は破棄される。

### A.4.4 start メソッド

before メソッドの処理が終わったジョブオブジェクトに対して適用される。引数はジョブオブジェクトへの参照である。複数のモジュールで start メソッドが定義されていた場合は、最も specialized なクラスの start メソッドが呼び出される。

メソッドの本体では、\$obj->NEXT::start() (\$obj は引数として渡されたジョブオブジェクトへの参照) のようにして親クラスの start メソッドを呼び出すことができる。

core モジュールでも start メソッドが定義されており、ここではジョブスクリプトの生成およびバッチスケジューラへのジョブ投入処理を行う。子クラスの start メソッドにより core モジュールの start メソッドが呼び出されないと、この処理が行われなくなるので注意すること。

### A.4.5 after\_isready メソッド

状態 (3.1 節) が done になったジョブオブジェクトに対して適用される。一般には、core::start メソッドによってバッチスケジューラに投入したジョブからのジョブ完了通知により、ジョブの状態が done になる。引数はジョブオブジェクトへの参照である。複数のモジュールで after\_isready メソッドが定義されていた場合は、親クラス 子クラスの順で全ての after\_isready メソッドが呼び出される。

各メソッドは、真偽値を返す。呼び出された after\_isready メソッド列のうち、1 つでも偽を返したメソッドがあれば、しばらくの時間待ち合わせした後、もう一度 after\_isready メソッド列の実行が行われる。

### A.4.6 after メソッド

after\_isready メソッド列が適用され、その全てが真を返したジョブオブジェクトに対して適用される。引数はジョブオブジェクトへの参照である。複数のモジュールで after メソッドが定義されていた場合は、親クラス 子クラスの順で全ての after メソッドが呼び出される。

メソッドの戻り値は破棄される。

### A.4.7 before\_isready, before, start, after\_isready, after の並行性

before, before\_isready, start, after, after\_isready の各メソッドは Xcrypt のユーザスレッドとは並行に実行される。ただし、各ジョブオブジェクト間についてのこれらのメソッドの実行は完全に並行に実行されるわけではない。Xcrypt 処理系が保証する並行性は以下の通りである。

- どのジョブオブジェクトの before\_isready, before, start, after\_isready, after メソッドも Xcrypt のユーザスレッドとは並行に実行される。
- どのジョブオブジェクトの before\_isready, before, start メソッドも 2 つ以上並行に実行されることはない。
- どのジョブオブジェクトの after\_isready, after, メソッドも 2 つ以上並行に実行されることはない。

- あるオブジェクトに対して `before_isready` メソッド列が全て真を返した際、それに続くそのオブジェクトへの `before` メソッド列の実行の前に、他のどのオブジェクトへの `before_isready` , `before` , `start` メソッド適用も行われることはない。
- あるオブジェクトへの `before` メソッド列の適用と `start` メソッドの適用の間に、他のどのオブジェクトへの `before_isready` , `before` , `start` メソッド適用も行われることはない。
- あるオブジェクトに対して `after_isready` メソッド列が全て真を返した際、それに続くそのオブジェクトへの `after` メソッド列の実行の前に、他のどのオブジェクトへの `after_isready` , `after` メソッド適用も行われることはない。