

シナリオ記述言語 *Q*

Kawa 版

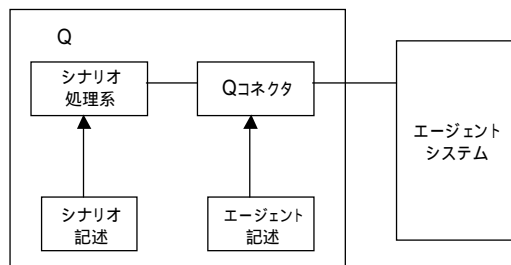
***Q* コネクタ仕様書**

v1.2

2004.2.12

京都大学情報学研究科石田研究室
連絡先：q@lab7.kuis.kyoto-u.ac.jp

1. 概要



構成図

Q コネクタは、 Q 処理系とレガシーなエージェントシステムを結びつけるための仲介役をはたすものである。仲介するためのルールは、提供される Scheme ベースの幾つかのマクロや関数を用いて記述する事が出来る。

Q 処理系から見ると、 Q コネクタは、エージェントそのものである。 Q 処理系はキュー・アクションの実行が必要になった場合は、 Q コネクタに処理を依頼し、その結果を取得する。 Q 処理系は、エージェント実装については、何も知らなくてよい。

Q コネクタは、 Q 処理系とエージェント実装間の仲介を行うものであるが、具体的には以下のような事を行う事が出来る。

エージェント記述内のキューやアクションのアダプタ記述を解釈し、レガシーなエージェントシステムが提供するメソッドを呼び出す事が出来る。

エージェント記述では、型変換処理、引数の順番の並び替え、戻り値の処理などを記述する事が出来る。

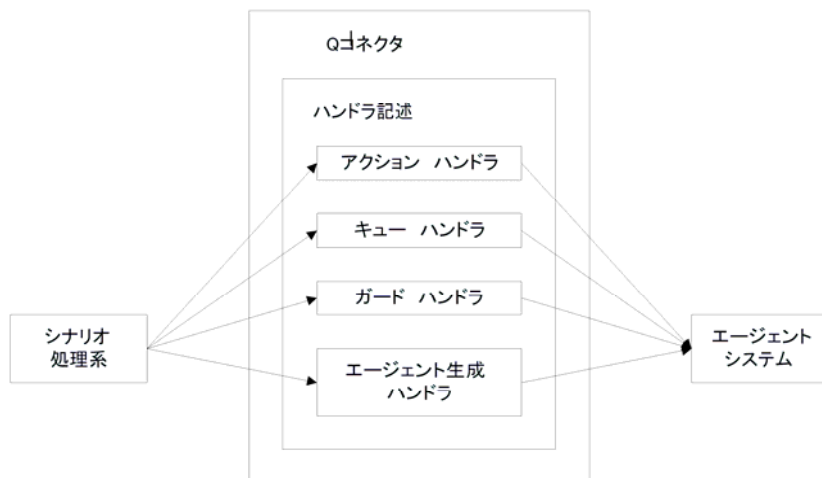
エージェント記述では、繰り返し、条件分岐、複数の処理の逐次実行などをスクリプティングする事が出来る。

通信機能を持たないエージェント間の簡単なイベント伝達、検索が出来る。

2. エージェント記述

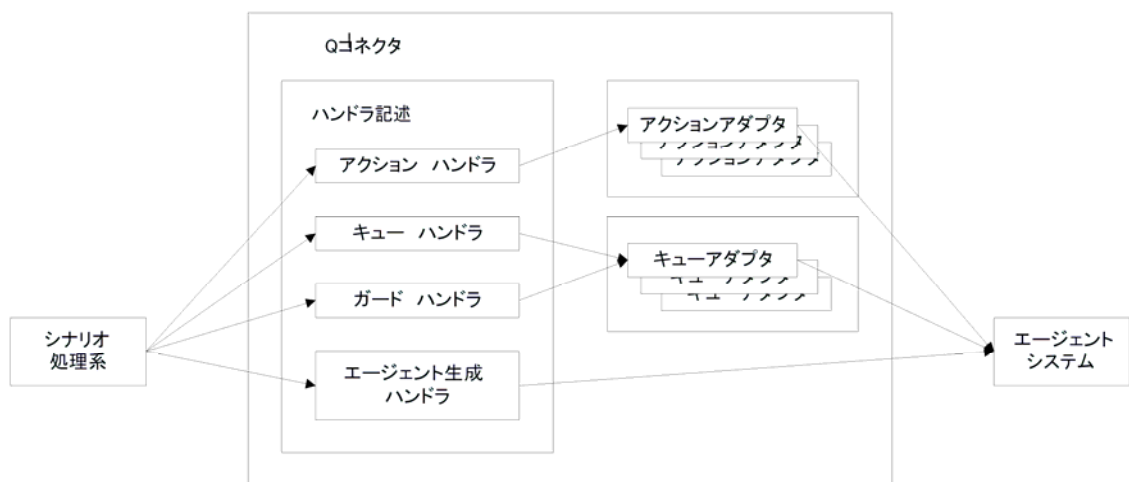
エージェント記述には2種類の実装方式をサポートしている。

2.1 ハンドラ記述のみの実装方法



シナリオ処理系から依頼された，アクション・キュー・ガードの実行要求を，そのままエージェントシステムに渡す仕組みである．エージェントシステムは，アクションの実行受付，キューの実行受付，ガードの実行受付といったインターフェースを公開している必要がある．

2.2 ハンドラ記述とアダプタ記述を組み合わせた実装方法



シナリオ処理系から依頼された，アクション・キュー・ガードの実行要求を，各ハンドラが受け取る．各ハンドラは，アクションやキューの種別に応じたアクションアダプタやキューアダプタを呼び出す．各アダプタには適切な処理手順が記述されており，対応するエージェントシステムのメソッドを呼び出す．エージェントシステムは，個別のアクション実行・キュー実行などのインターフェース，もしくは，より低レベルなインターフェースを公開している必要がある．

3. ハンドラ記述

ハンドラにはアクションハンドラ，キューハンドラ，ガードハンドラ，エージェント生成ハンドラがある．それぞれ，以下のように定義する事が出来る．

3.1 defguardhandler

シナリオ処理系からガードの実行要求が来たときに呼ばれるハンドラであり，その時の処理を記述する．

エージェントシステム上に，任意のガード処理を受け付けるインターフェースが公開されている場合は，型変換などの整形処理を施してから，エージェントシステム上のインターフェースを呼び出せば良い．

もし，各キューアダプタを後述の方式に従って定義している場合は，このハンドラ定義は省略する事も出来る．

形式は以下の通り．

```
(defguardhandler (agent cue-list)
  ...)
```

agent には，ガードを実行したエージェントが渡される．

cue-list には並行観測すべきキューのリストが渡される．

選択されたキューのインデックス(一番目の要素なら 0)を返す．

この定義がなかった場合，以下のように振舞う．select-cue 関数は，キューアダプタを順番に呼び出すことにより，擬似的な並行観測をするものであり，選択されたキューのインデックスを返す．

```
(defguardhandler (agent cue-list)
  (select-cue agent cue-list))
```

3.2 defcuehandler

シナリオ処理系からキューの実行要求が来たときに呼ばれるハンドラであり，その時の処理を記述する．

エージェントシステム上に，任意のキュー処理を受け付けるインターフェースが公開されている場合は，型変換などの整形処理を施してから，エージェントシステム上のインター

フェースを呼び出せば良い．

形式は以下の通り．

```
(defcuehandler (agent cue-name args)
  ...)
```

cue-name にはキューの名前，args にはキューの引数が渡される．

この定義がなかった場合，以下のように振舞う．run-cue 関数は，defcueadapter で定義されたキューを呼び出すものである．

```
(defcuehandler (agent cue-name args)
  (run-cue cue-name agent args))
```

3.3 defactionhandler

シナリオ処理系からアクションの実行要求が来たときに呼ばれるハンドラであり，その時の処理を記述する．

エージェントシステム上に，任意のアクション処理を受け付けるインターフェースが公開されている場合は，型変換などの整形処理を施してから，エージェントシステム上のインターフェースを呼び出せば良い．

形式は以下の通り．

```
(defactionhandler (agent action-name args)
  ...)
```

action-name にはアクションの名前，args にはアクションの引数が渡される．

この定義がなかった場合，以下のように振舞う．run-action 関数は，defactionadapter で定義されたアクションを呼び出すものである．

```
(defactionhandler (agent action-name args)
  (run-action action-name agent args))
```

3.4 defagenthandler

シナリオ処理系からエージェントの生成要求が来たときに呼ばれるハンドラである。通常は、型変換などの整形処理を施してから、エージェントシステム上のインターフェースを呼び出せば良い。必要なら、その他の処理をここに記述する事も出来る。

形式は以下の通り。

```
(defagenthandler (name args agent)
  ...)
```

name にはエージェント名, args にはエージェントの引数, agent にはエージェントオブジェクトが渡される。

例えば, Q 処理系側で,

```
(defagent Agent
  :agentmanager "rmi://localhost/am0"
  :foo "foo"
  :bar 3)
```

と実行した場合,

```
name <-- "Agent"
args <-- '(:agentmanager "rmi://localhost/am0"
:foo "foo"
:bar 3)
agent <-- agent object
```

が割当てられ, defagentbody の中身が実行される。

args のキーワード引数を取得するには,

```
(get-keyword-arg :keyword args default-value)
```

と呼出す。もし args に :keyword 引数が指定されていればその値を返し, 指定されてい

ければ default-value を返す .

上記の例の場合 ,

```
(get-keyword-args :foo args #f)
```

とすれば , "foo" を取得する事が出来る .

注 : この関数は即座に終了する必要がある . もしエージェントの処理をバックグラウンドで実行させ続けたい場合は , KAWA の future 機能を用いる必要がある .

4. アダプタ記述

キュー・アクションのインターフェース宣言は , シナリオ内で , defcue, defaction を用いて記述されるが , Q コネクタでは , 対応するキュー・アクションのアダプタ記述を以下のように記述する事が出来る .

```
(defcueadapter name &key (key val) ... &pattern (pattern val) ...)  
  (defactionadapter name &key (key val) ... &pattern (pattern val) ...)
```

引数にキーワードを受け付ける為には &key を用いる . パターン変数を受け付ける場合は , &pattern を用いる .

例えば ,

```
(defcueadapter ?hear (&key (from #f)  
                      &pattern (message ""))  
  ...)  
  
(defcueadapter !speak (&key (to #f)  
                      &pattern (message ""))  
  ...)
```

と記述する事が出来る .

4.1 キーワード変数の扱い

以下のようなシナリオが実行されると、!speak の呼び出しが発生するが、

```
(defscenario scenario ()
  (scene1
    (otherwise
      (!speak :to agent :msg "hello!")))))
```

Q 処理系は対応する Q コネクタの!speak のアダプタ記述に基づいて対応するエージェントのメソッドを呼び出す。

```
(defactionadapter !speak (&key (to #f) (msg ""))
  ...
  (display msg)
  ...)
```

この時、シナリオに書かれた、!speak のキーワード引数(:to, :msg)は、それぞれ、defactionadapter の引数宣言部に書かれた、to, msg にマッピングされる。

4.2 パターン変数の扱い

パターン変数の値は、参照値である。参照値は実体値をカプセル化した値である。参照値の指す参照先の実体は取得・変更する事が出来るので、主に、キュー・アクションの内部で、値の書き換えが発生する場合に用いられる。

パターン変数を利用するには、必ず、&pattern 宣言を行わなければならない。例えば、?hear のアダプタ記述において、message 値は out 属性を持っていたとする。そのような場合は、message 引数は &pattern 宣言する必要がある。たとえば、

```
(defcueadapter ?hear (&key (from #f)
  &pattern (message ""))
  ...)
```

と定義すれば、form はキーワード変数、message はパターン変数として扱われる。

パターン変数を扱う為に、以下の 3 つの API が用意されている。

1. refval REF
2. set-ref! REF VALUE
3. make-ref VALUE

refval は、参照 REF の参照先の値を取得する。

set-ref は、参照 REF の参照先の値を VALUE に更新する。

make-ref は、VALUE を参照する参照値を作成する。

例えば、以下のように用いる。

```
(defcueadapter ?hear (&key (from #f)
                          &pattern (message ""))
  ...
  (refval message)
  ...
  (set-ref! message "Hello!")
  ...
)

(defscenario scenario (&pattern ($message ""))
  (scene1
    ((?hear :from Taro :message $message)
     (display (refval $message))))))
```

defscenario のパターン宣言部において \$message パターン変数が空文字列に初期化される。scene1 の ?hear の呼出しで、?hear のアダプタ記述に処理が移る。その時、message には、空文字列への参照が入っている。この時、(refval message) で参照先の値を取得する事ができ、(set-ref! message "Hello!") で参照先の値を更新する事が出来る。?hear のアダプタ記述を抜けてから、

```
(display (refval $message))
```

を実行すると、?hear 内で設定した "Hello!" が表示される。

4.3 キューの並行観測

キューアダプタを用いた、ガードコマンドによる並行観測は、単一スレッド上で、複数のキューアダプタの呼び出しを順番に行う事によって、擬似的な並行観測として実現してい

る。(注: 擬似的な並行観測を行いたくない場合は, 前述の `defguardhandler` を用いて, エージェントシステム上のガード受付インターフェースを直接呼び出す事が出来る)

キューの実装は, 必ず, `let-polling` マクロを用いて記述する必要がある。 `let-polling` マクロの形式は以下の通り。

```
(let-polling ((var_1 val_1)
             (var_2 val_2)
             ...
             (var_n val_n))
  exp_1
  exp_2
  ...
  exp_m)
```

動作は以下の通り。

1. 各 `var_i` ($1 \leq i \leq n$) に `val_i` を評価した値を代入する。
2. すべての `var_i` ($1 \leq i \leq n$) が真でないなら, 一定の時間 `sleep` し, 1 に戻る。
3. すべての `var_i` ($1 \leq i \leq n$) が真なら, `exp_k` ($1 \leq k \leq m$) を逐次実行する。

例えば, `?hear` の実装は以下のようなになる。(search-event については後述)

```
(defcueadapter ?hear (self
                     &key (from #f)
                     &pattern (message ""))
  ;; イベントテーブルから,
  ;; ・event-type が speak で,
  ;; ・speak-to が自分であるような,
  ;; イベントをポーリングする。
  (let-polling
    ((event
      (search-event
        (%and (%equal event-type 'speak)
```

```
(%equal speak-to self))))))
;; 合致するイベントがあったら，キューが成立．
;; この時，event には合致したイベントが設定されている．
;; 参照 message に適切な値を設定し，終了．
(set-ref! message (get-event-arg event 'message))))
```

4.4 イベント

エージェント間のメッセージ交換にはイベントを用いる．
イベントは，任意個の属性を持つ．エージェント実装間で合意があれば，任意のイベントを用意する事が出来る．

```
EVENT = ((ATTR-NAME ATTR-VAL) ...)  
ATTR-NAME = シンボル  
ATTR-VAL = オブジェクト
```

属性は，属性名と属性値で示される．
属性名はシンボルで，属性値は任意の Scheme オブジェクトである．
予約されている属性名として，event-type, event-id がある．

event-type はイベント名を示すものである．エージェント実装間で合意があれば，イベント名は任意のシンボルが利用出来る．

event-id は，イベントが登録される度に自動的に割り振られる，イベントに一意的な ID である．

4.5 イベント処理

defcuedapter, defactionadapter 内で，イベント処理を行う事が出来る．
通信機能を持たないエージェントやオブジェクトに，イベント処理を組み込むことで，エージェント間の連携が可能になる．

```
(register-event-peer agent event-type expire . event-args)
```

register-event-peer はイベントを各エージェントが個別に持っているイベントテーブルに登

録する .

event-type にはイベント名を指定する .
expire にはイベントの有効期間(秒)を指定する .
event-args には任意個の名前付き引数を指定する .

例えば ,Refrigerator から Air-Conditioner に "Hello" としゃべった事を示すイベントを登録し , 60 秒間 , 有効にしたい場合は , 以下のように記述する . この場合 , 60 秒が経過すると , テーブルから自動的に削除される .

```
(register-event-peer
  Refrigerator
  'speak
  60
  (cons 'speak-from Refrigerator)
  (cons 'speak-to Air-Conditioner)
  (cons 'message "Hello"))
```

```
(search-event-peer query agent)
```

search-event-peer は条件の一致するイベント構造をエージェントに個別のイベントテーブルから検索する .

条件の一致するイベントが見付からない場合は #f を返す .
query には query language に従った構造を指定する .
agent には検索者(自分自身)を示すエージェントオブジェクトを指定する .

例えば , Taro からの speak イベントを取得したい場合は ,

```
(search-event-peer
  (%and (%equal event-type 'speak)
        (%equal speak-from Taro)))
  self)
```

で取得する事が出来る .

query language の仕様は以下の通り .

(%and QUERY ...)

すべての QUERY が成立した時のみ真 .

(%or QUERY ...)

すべての QUERY が成立しなかった時のみ偽 .

(%equal TYPE-NAME VALUE)

TYPE-NAME の値が 「equal?」 関係にあった時のみ真 .

(%= TYPE-NAME VALUE)

TYPE-NAME の値が 「=」 関係にあった時のみ真 .

(%> TYPE-NAME VALUE)

TYPE-NAME の値が 「>」 関係にあった時のみ真 .

(%< TYPE-NAME VALUE)

TYPE-NAME の値が 「<」 関係にあった時のみ真 .

%true

常に真 .

%false

常に偽 .

(%not QUERY)

QUERY が 偽の時のみ真 .

(consume-event-peer id agent)

イベントの消費フラグを設定する .

consume-event-peer は , エージェントに個別のイベントテーブル上の指定されたイベントに対して消費フラグを設定する . ここで , id とは , 前述の event-id である .

イベントの消費フラグが設定されたイベントは，`search-event-peer` で検索されない．逆に消費フラグを設定しないと何度も検索されてしまう．

```
(get-event-arg event name)
```

名前を指定し，イベント構造から値を取得する．

例えば，`search-event` で取得したイベント構造から，`'message` 値を取得したい場合は，

```
(get-event-arg event 'message)
```

で取得する事が出来る．もし，`'message` 値が存在しない場合は `#f` が返る．

イベントの ID は，

```
(get-event-arg event 'event-id)
```

で取得する事が出来る．同様にイベント名は，

```
(get-event-arg event 'event-type)
```

で取得する事が出来る．

4.6 エージェントの属性

```
(set-attribute! agent attribute-sym value)
```

指定されたエージェントの属性名 `attribute-sym` に属性値 `value` を設定する．

```
(get-attribute agent attribute-sym)
```

指定されたエージェントの属性名 `attribute-sym` の値を取得する．

```
(get-agent-name agent)
```

エージェントの名前(文字列)を取得する .

5. 補足

5.1 Java との連携

Java で記述したコードと連携したい場合は ,KAWA が提供する `invoke` 命令などを用いる .

```
(make <JavaClassName> arg ...)
```

クラス `JavaClassName` のオブジェクトを生成する .

```
(invoke JavaObject 'MethodName arg ...)
```

`JavaObject` の オブジェクトメソッド `MethodName` を呼出す .

```
(invoke-static <JavaClassName> 'MethodName arg ...)
```

`JavaClassName` の クラスメソッド `MethodName` を呼出す .

詳しくは ,

<http://www.gnu.org/software/kawa/>

を参照の事 .