Coval 0.81 マニュアル

Version 0.8.02　2013-04-05 荻原剛志

1　はじめに

1.1　対象とするバージョン

本稿は文献 [1] で提案した、C 言語で利用可能なデータバインディング用ライブラリである coval について、その具体的な利用方法を記述したものである。対象とするバージョンは 0.81(Apr. 2013) である。

本稿では、基本的な coval の機能について述べ、次に coval bridge などの拡張機能について説明する。拡張機能の部分は coval の基本機能からの独立性が高く、今後、存廃も含めた仕様変更が行われる可能性がある。

1.2　coval の概要

coval は、モジュール間で値を共有する仕組みを備えた一種の変数と考えることができ、値を格納、参照することができる。coval の実体は構造体として実装されており、以下ではこの構造体を coval 構造体と呼ぶことにする。

coval は同じ型のデータを保持する他の coval と値を共有するようにできる。この操作をバインドと呼ぶ。バインドは２個以上の複数個の coval の間で行うことができる。バインドされたすべての coval は共通の変数を参照しているため、いずれかの coval で値を更新すると他のすべての coval の値も変化したように見える。

coval にはコールバック関数を対応させておくことができ、バインド中に共有変数に対して更新の操作が行われると自動的に呼び出される。更新の前後で値が変化しなくても呼び出しは発生する。関数を指定しなければ呼び出しは発生しない。

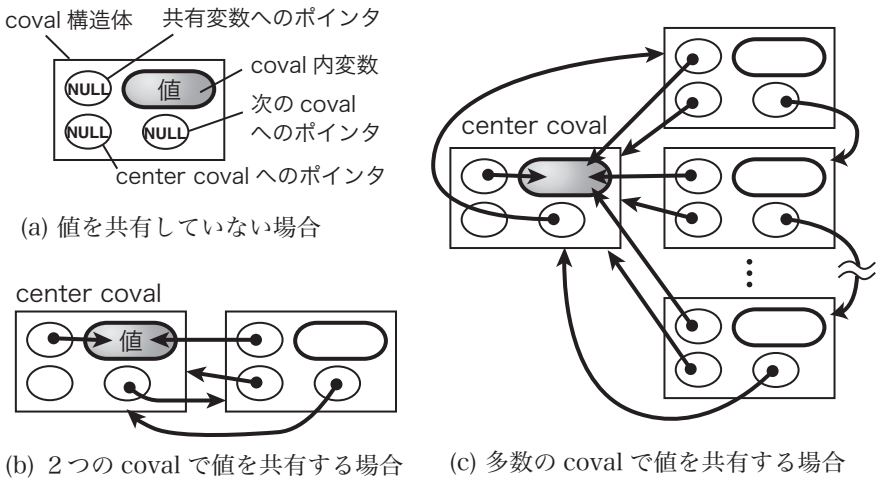


図 1: coval 構造体とバインドの実現

coval 構造体の概念を図 1(a) に示す。構造体はバインドに必要な情報のほか、起動される関数へのポインタなども含む。coval は、任意の型の値を扱えるように、マクロとライブラリを組み合わせて実装されている。図 1(b) は２つの coval 構造体がバインドによって値を共有する場合、(c) は複数の coval 構造体がバインドされている場合を表す。

1.3 coval をアプリケーションで利用するには

coval は組込みシステムでの利用も想定しており、C 言語の標準関数も含め、他のライブラリは一切利用しない。メモリ領域の動的な確保も行わない。

coval 0.81 の実装は、以下のファイルから成る。最小構成は coval.h と coval.c である。

coval.h

coval の各種宣言やマクロが記述されたヘッダファイルである。coval を利用するソースコードで include する必要がある。

下記の coval bridge を利用しない場合、coval.h 内で定義されているマクロ `USE_COVAL_BRIDGE` を 0 としてコンパイルする必要がある。gcc であれば、ソースコードを書き直す必要はなく、コンパイル時にオプション `-DUSE_COVAL_BRIDGE=0` を指定すればよい。

coval.c

coval を構成する実行コードが記述されている。アプリケーションプログラムにリンクする必要がある。

covalbridge.h

coval bridge（4 章を参照）の各種宣言やマクロが記述されたヘッダファイルである。coval bridge を利用する場合、そのソースコードで include する必要がある。coval bridge を利用しない場合には必要ない。

covalbridge.c

coval bridge を構成する実行コードが記述されている。coval bridge を利用する場合にはアプリケーションプログラムにリンクする必要があるが、利用しないのであればアプリケーションに含める必要はない。

covalsymbol.*

コンパイル後、識別子（文字列）によって coval を指定する試みのためのサンプルコードである。本稿で述べる範囲の coval の用途では必要ない。

2 coval のマクロ

2.1 coval 構造体の型

coval では、共有される値の型をマクロの引数とすることによって、コンパイル時に複数の構造体を自動的に構築する。

型 `T` のデータを共有変数として内部に含む coval 構造体の型は、`CovalTypeOf(T)` と表わされる。これを、型 T を共有型とする coval 構造体型と呼ぶことにする。本稿の以下の説明で、マクロ（あるいは関数）の引数の型として「coval 構造体」と記述した場合、マクロ `CovalTypeOf` を使って宣言した型を指すものとする。

例えば、int 型の値を共有するための coval を変数 a に作る場合は以下のように宣言できる。

```
CovalTypeOf(int) a;
```

あるいは、typedef 宣言を用いて以下のようにすることもできる。

```
typedef CovalTypeOf(int) covalInt;
covalInt a;
```

参考

マクロ `CovalTypeOf` を使って宣言する構造体は、共有されるデータ型によってそれぞれに異なるが、それらの共通部分のデータ型は coval の実装内で `coval_base` 型として参照される。ただし、アプリケーション側のプログラムでは、`coval_base` 型を明示的に使用することは避け、実際に存在する coval の型を使用するべきである。この `coval_base` 型は実装の都合で使用している宣言上の存在であり、実際のプログラム内には存在しないデータだからである。`coval_base` 型は、実体を持つ coval 構造体よりもオブジェクトのサイズが小さいため、型として指定すると実行時にエラーが発生する可能性がある。

2.2 coval 変数の初期値

coval 構造体が保持する共有値の初期値は、他の C 言語の変数と同様に、静的変数の場合には初期値が 0、あるいはポインタとしては NULL ポインタが値となる。また、バインドはされていない状態、コールバック関数も指定がない状態が初期値となる。

一方、動的に獲得したメモリ上に作られた coval 構造体は、不定な値を持つ可能性がある。このような変数を、バインドされていない状態、コールバック関数の指定がない状態に初期化するため、マクロ `CovalInit` を利用することができる。ここで、cov はマクロ `CovalTypeOf` を使って宣言した coval 構造体へのポインタ型とする。

```
CovalInit(cov a);
```

このマクロは不定な値を持つ構造体に最低限の初期値を設定するだけである。静的領域に確保された変数に対して、このマクロを適用する必要はない。coval の共有値については何の設定も行わない。また、バインドしている coval をアンバインドする機能を持っているわけではないことにも注意が必要である。

なお、動的に獲得したメモリに coval 構造体を置く場合、アンバインドせずにメモリを解放したりしないように注意が必要である。

参考

coval 構造体を自動変数として利用することは推奨しない。バインドされたままの coval 変数がスタックからなくなってしまった場合に、誤動作が発生するからである。また、モジュール間で値を共有するという coval の基本的な働きを考えた場合、coval を自動変数に設定することにはあまり意義を見出すことができない。利用不可能ではないが、利用中に変数が存在し続けることをプログラマの責任で保証する必要がある。

2.3 バインド

coval 同士のバインドはマクロ `CovalBind` を用いて、プログラムの実行中に動的に行う。ここで、cov はマクロ `CovalTypeOf` を使って宣言した coval 構造体へのポインタ型とする。

```
CovalBind(cov *c1, cov *c2);
```

例えば、double 型を共有する coval 構造体が２つの変数 a, b に格納されている場合、以下のようにしてバインドすることができる。

```
CovalTypeOf(double) a;
CovalTypeOf(double) b;

CovalBind(&a, &b);
```

マクロ `CovalBind` の第 1 引数の coval はすでにバインドされていてもよいが、第２引数にはまだバインドされていない coval を指定する。バインド後に共有される値は、第 1 引数が保持、または共有している値になる。

共有型の異なる coval 同士をバインドしてはならない。

バインドされた coval の集合を coval グループと呼ぶ。coval グループ内には、共有値を保持し、動作を管理するための coval が 1 つ存在し、center coval と呼ぶ。図 1(c) は複数の coval が center coval の保持する変数を共有する概念を示す。ただし、coval へのアクセスにはすべてマクロを用いるため、center coval の存在を意識してプログラミングする必要はない。

2.4 アンバインド

バインドされた複数の coval から、特定の coval を指定して共有関係から取り除く（アンバインドする）ことができる。このとき、指定した coval は center coval であってもよく、プログラム側からは center coval かどうかを意識する必要はない。

アンバインドにはマクロ `CovalUnbind` を使う。ここで cov は coval 構造体へのポインタ型とする。

```
CovalUnbind(cov *c);
```

2.5 コールバック関数の指定

coval にコールバック関数を設定するには、マクロ `CovalSetCallback` を使う。coval 変数 a にコールバック関数 fn を設定するには次のようにする。引数 ptr は任意のポインタとする。

```
CovalSetCallback(&a, fn, ptr);
```

コールバック関数は次の形式で定義されている。第 1 引数、第２引数には、それぞれ、この関数を指定した coval 変数へのポインタ、および `CovalSetCallback` で指定した任意のポインタ（上記の例では `&a` および `ptr`）が渡される。

```
int 関数名 ( cov *, void * )
```

ただし、実際に関数を定義する場合には、具体的な coval 構造体のポインタ型と、実際のポインタ型を使うべきである。void *型を使う必要はない。

関数の返り値は、処理に成功した場合にマクロ COVAL_PROC_SUCCESS で定義された値を、処理に失敗した場合にマクロ COVAL_PROC_ERROR で定義された値を返すようにする（節 3.1 を参照）。

コールバック関数を指定するとき、マクロ CovalSetCallback の第 2 引数として関数の代わりに NULL ポインタを渡すと、コールバック関数を使わない指定となる。

マクロ CovalSetCallback の第 3 引数として任意のポインタを指定できる。このポインタを利用すると、複数の coval に対して同じコールバック関数を指定しても、呼び出された際の動作を変えることができる。

コールバック関数は、バインドするよりも前に coval に設定することができる。また、バインドしているかどうかに関わらず、実行中にいつでも設定を変更できる。

例えば、float 型を共有型とする coval 変数 a と b があり、coval の値が変更された時、float 型の値を四捨五入した整数値を変数 m と n に代入したいとする。このような場合、以下のように記述することができるだろう（マクロ CovalValue は coval の値を参照する。節 2.6）。

```
typedef CovalTypeOf(float) covFloat;
covFloat a, b;

static int roundOff( covFloat *cp, int *valp )
{
    if (valp)
        *valp = (int)(CovalValue(cp) + 0.5);
    return COVAL_PROC_SUCCESS;
}

...
    int m, n;

    CovalSetCallback(&a, roundOff, &m);
    CovalSetCallback(&b, roundOff, &n);
```

2.6 値の参照と更新

coval 内の共有値の参照、更新はマクロを介して行う。個々の coval はマクロ CovalTypeOf で宣言した共有型を持ち、値に対する操作に関してはコンパイル時に型チェックが行われる。

値を更新するマクロは CovalAssign である。第 1 引数に coval 構造体へのポインタ、第 2 引数に代入すべき値を指定する。

```
CovalAssign( cov a, 式 );
```

共有変数の値を更新した後、coval グループ内の coval にコールバック関数があれば、それらを順番に起動する。ただし、CovalAssign の引数となった coval 自体に設定されている関数は呼び出さない。coval がバインドされていない場合、その coval の持つ値が更新されるだけである。

coval グループ内に複数個のコールバック関数が設定されていた場合、呼び出される順序は、その coval がバインドされた順になる。

次の例では、int 型を共有型とする coval に式の値を代入している。

```
CovalTypeOf(int) a;

CovalAssign(&a, (int)(days / 7));
```

値を参照するマクロ CovalValue は、coval がバインドされていれば共有変数の値を返し、バインドされていなければ、その coval の持つ値を返す。次の記法で、構文内で式として使用できる（引数は coval 構造体のポインタ）。

```
CovalValue( cov a )
```

次の例では、unsigned int 型を共有型とする coval の値を式の中で参照している。

```
CovalTypeOf(unsigned int) u;

int flag = (CovalValue(&u) & 0x0080) != 0;
```

2.7 コールバック関数の起動を伴う値の更新

上で述べたマクロ CovalAssign は、バインドしている coval に対して適用した場合、その coval グループ内で、マクロの引数以外の coval に対してコールバック関数を起動するという動作をする。一方、共有値の更新の元となった coval に対してもコールバック関数を起動すべき場合がある。そのため、共有値の変更のために、もうひとつのマクロ CovalSetValue が用意されている。使い方は CovalAssign と同じである。

```
CovalSetValue( cov, 式 );
```

マクロ CovalSetValue は、指定した coval がバインドされていた場合、その coval グループ内で、マクロの引数である coval も含むすべての coval についてコールバック関数を起動する。さらに、指定した coval がバインドされていなくても、値の変更に伴ってコールバック関数を起動する。

次の例で、(1), (2) の位置では coval 変数 a にはコールバック関数が設定されているが、まだ他の coval とバインドされてはいない。ここでマクロ CovalSetValue を使って値を設定すると、関数 funcA が起動される。代わりに CovalAssign を使った場合、関数は起動されない。

次に (3), (4) の位置では a, b, および c は互いにバインドされて coval グループを構成している。このとき、マクロ CovalSetValue で共有値を更新すると、関数 funcA, funcB, funcC のすべてが起動される。一方、マクロ CovalAssign を使った場合には、関数 funcA, funcB は起動されるが関数 funcC は起動されない。

```
CovalTypeOf(int) a, b, c;

CovalSetCallback(&a, funcA, NULL);
CovalSetCallback(&b, funcB, NULL);
CovalSetCallback(&c, funcC, NULL);

CovalSetValue(&a, 0);  // (1) funcA が起動される
CovalAssign(&a, 1);    // (2) 関数は起動されない
```

```
CovalBind(&a, &b);      // ここでバインドする
CovalBind(&a, &c);
CovalSetValue(&c, 10); // (3) funcA, funcB, funcC が起動される
CovalAssign(&c, 10);   // (4) funcA, funcB が起動される
```

2.8 構造体を共有型とする場合

coval の共有型として、整数や実数のような単純型だけではなく、構造体を指定することもできる。このような場合、構造体のメンバを指定して値の参照や更新を行う必要がある。

マクロ CovalMemberValue、マクロ CovalAssignMember は、構造体のメンバを指定して値の参照、および更新ができる。マクロの引数は以下の通り。

```
CovalMemberValue( cov, メンバ )         // 値を参照
CovalAssignMember( cov, メンバ, 式 )    // 値を更新
```

マクロの第２引数にはメンバ名を記述する。この部分は式を記述することができない。

次の例では、struct point 型を共有型とする coval について、メンバを指定して値の参照や更新を行う例である。

```
struct point {
    float x, y;
};
CovalTypeOf(struct point) a;

struct point p1 = { 10.0, 12.0 }, p2;
CovalAssign(&a, p1);                  // (1) 構造体ごと代入

float f = CovalMemberValue(&a, x);    // (2) メンバ x の値を参照
CovalAssignMember(&a, y, f * 2);      // (3) メンバ y の値を更新
```

コールバック関数の呼び出し方に関しては、マクロ CovalMemberValue と CovalAssignMember は、節 2.6 で扱ったマクロ CovalValue と CovalAssign と同様である。つまり、coval がバインドしていない場合はコールバック関数を起動せず、バインドしている場合でもマクロの引数である coval が持つコールバック関数は起動しない。

これに対して、メンバを指定して値を更新する際、節 2.7 で扱ったマクロ CovalSetValue のように振る舞うマクロとして CovalSetMemberValue が用意されている。

```
CovalSetMemberValue( cov, メンバ, 式 )    // 値を更新
```

構造体の値を更新する場合、いくつかのメンバの値を同時に変更したい場合がある。すべての値を更新するには、上記の例のようにマクロ CovalAssign を使って構造体をまるごと代入する方法が利用できる。しかし、メンバのうちの一部分だけを変更したい場合、ひとつひとつの値に対してマクロ CovalAssignMember や CovalSetMemberValue を使うと、コールバック関数が必要以上に呼び出されてしまい、パフォーマンスの低下につながる可能性がある。

以下のマクロは、複数個のメンバ（M1, M2, ...）に対してひとつずつ値 (式 1, 式 2, ...) を設定し、最後にコールバック関数を 1 回だけ起動する。コールバック関数の呼び出し方は、マクロ CovalAssignMembers2 他はマクロ CovalAssignMember と同様、マクロ CovalSetMemberValue2 他はマクロ CovalSetMemberValue と同様である。

```
CovalAssignMembers2(cov, M1, 式 1, M2, 式 2)
CovalAssignMembers3(cov, M1, 式 1, M2, 式 2, M3, 式 3)
CovalAssignMembers4(cov, M1, 式 1, M2, 式 2, M3, 式 3, M4, 式 4)

CovalSetMemberValue2(cov, M1, 式 1, M2, 式 2)
CovalSetMemberValue3(cov, M1, 式 1, M2, 式 2, M3, 式 3)
CovalSetMemberValue4(cov, M1, 式 1, M2, 式 2, M3, 式 3, M4, 式 4)
```

参考

メンバ名の引数は、記述された通りにマクロ展開されるだけであるため、構造体がメンバとして構造体を含む場合、次のような書き方ができる。

```
struct lineSegment {
    struct point start, end;
};
CovalTypeOf(struct lineSegment) a;

CovalAssignMember( &a, start.x, 0.0 );
CovalAssignMember( &a, end.y, 3.14 );
```

C の共有型についても、同様に記述できる。また、“.” 演算子ではなく、“->” 演算子を利用することも可能である。ただし、過度な利用はプログラムの分かりやすさとモジュールの独立性を損なう可能性があるので注意が必要である。

参考

coval の共有型として構造体を指定することはできるが、配列を指定することはできない。

3 coval 0.81 の拡張機能

前章では、論文で公表している coval の基本的な機能について述べた[1]。

この章では、coval 0.81 に実装されているその他の付加的な機能について述べる。これらの機能は試験的な拡張機能であり、将来の coval の仕様に採用されるとは限らない。

3.1 処理が成功したらコールバック関数の呼び出しをやめる

節 2.6 において、coval グループ内の coval でコールバック関数を設定しているものが複数あった場合、共有値の更新によってそれらが順番に呼び出されると説明した。

一方、共有値の更新に伴い、すべてのコールバック関数を 1 回ずつ呼び出すのではなく、いずれかのコールバック関数によって何らかの条件が満たされた時点で処理を打ち切りたい場合も存在する。例え

[1] 節 3.1 の機能は文献 [2] に概要を示している。

ば、画面上にいくつかのボタンが重ならずに配置されている状態で、画面がタッチされたとき、その座標を含むボタンが押されたとみなし、対応する処理を行わせたいとする。このような場合、該当するボタンが発見されたあとは他のボタンについての処理を行う必要はない。

コールバック関数の呼び出しにおいてこのような振る舞いをさせたい場合、その coval グループに対して BindingLoopBreakable 属性を設定する。設定には次のマクロを使う。ここで、cov は coval 構造体へのポインタ型であり、目的の coval グループに属する coval のうちのどれか 1 つを指定すればよい。属性の設定は coval グループに対してのみ有効であり、バインドされる前の coval に対して設定しておくことはできない。引数 flag は真偽値で、真の場合に BindingLoopBreakable 属性が設定され、偽の場合は属性が解除される。既定値では偽である。

```
CovalSetBindingLoopBreakable(cov a, int flag);
```

節 2.5 において、コールバック関数の返り値について簡単に説明したが、coval 0.81 では次の 3 つの返り値が用意されている。

COVAL_PROC_SUCCESS	処理に成功したことを示す
COVAL_PROC_IGNORED	実効性のある処理が行われなかった
COVAL_PROC_ERROR	処理に失敗した（エラーが発生した）

BindingLoopBreakable 属性を設定した coval グループの共有値が更新されてコールバック関数の呼び出しが発生した場合、いずれかのコールバック関数が値として COVAL_PROC_SUCCESS を返すと、それ以上、別のコールバック関数は呼び出されない。

上で述べたボタンの例であれば、座標値がボタンの領域に含まれる場合にはタッチに対応する処理を行ってから COVAL_PROC_SUCCESS を返し、領域に含まれない場合には COVAL_PROC_IGNORED を返すようにすればよい。

処理に失敗した場合の値として COVAL_PROC_IGNORED と COVAL_PROC_ERROR を用意しているが、現在の実装ではこれらは区別していない。ただし、意味の上からは、COVAL_PROC_IGNORED は正常な処理の一環であり、COVAL_PROC_ERROR は本来の意図と異なるエラーが発生したことを示す。

3.2 coval グループから新しい共有値を得る

これまで述べた coval の動作は、共有値が更新されたことを coval グループ内に一斉に知らせるものであった。いわば Push 型の動作と言える。

一方、共有値を更新するため、coval グループ内に更新のリクエストを送るという使い方も考えられる。２つの coval がバインドされている状態では、一方が他方に更新を依頼するかたちになる。更新のリクエストに応じて共有値を更新することができる coval は、コールバック関数とは別に、更新関数を持つものとする。

coval 変数 a に更新関数 fn を設定するにはマクロ CovalSetUpdate を使う。引数 ptr は任意のポインタとする。引数の coval はバインドされていても、いなくてもよい。

```
CovalSetUpdate(&a, fn, ptr);
```

更新関数は次の形式で定義されている。第 1 引数にはこの関数を指定した coval 変数へのポインタ、第２引数には CovalSetUpdate で指定した任意のポインタ、第 3 引数には更新すべき共有変数へのポインタが渡される。関数の返り値は、現在の実装では利用されていないが、COVAL_PROC_SUCCESS を返すように記述するのがよい。

```
int 関数名 ( cov *, void * , void * )
```

ただし、実際に関数を定義する場合には、具体的な coval 構造体のポインタ型と、実際のデータへのポインタ型を使うべきである。例えば、float 型を共有型とする coval 型を持つ変数 a に対し、int 型の変数（複数個の coval を識別するため）へのポインタを第２引数とする更新関数 funcR を設定するには次のようにすればよい。void *型を使う必要はない。

```
typedef CovalTypeOf(float) covalFloat;
covalFloat a;
int tag = 0;

int funcR(covalFloat *cp, const int *tagp, float *pp)
{
    float newvalue;
    if (*tagp == 0) { ... /* 何らかの計算 */... }
    ...
    *pp = newvalue;
    return COVAL_PROC_SUCCESS;
}

    ...
    CovalSetUpdate(&a, funcR, &tag);
```

coval グループに対して共有値の更新を求めるには、次のマクロ CovalGetValue を用いる。次の記法で、更新後の値を表す式として使用できる。引数 a は coval グループに属するいずれかの coval へのポインタである。

```
CovalGetValue( cov a )
```

引数の coval がバインドされていない場合、自分自身に更新関数が設定されていればその関数が呼び出されて値を更新する。coval グループに更新関数がない場合、およびバインドされておらず自分自身にも更新関数がない場合には現在の共有値の値が返される。

共有値が構造体である場合、更新した結果のうち、特定のメンバの値のみを参照したいことがある。そのような場合、マクロ CovalGetMemberValue を用いてメンバを指定すればよい。次の記法で、構文内で式として使用できる。なお、値の更新はそのメンバだけではなく、共有値の全体におよぶことに注意。

```
CovalGetMemberValue( cov a, メンバ )
```

バインドされていない場合、および更新関数が設定されていない場合の動作はマクロ CovalGetValue と同様である。

議論

３つ以上の coval がバインドされている状況で値の更新をリクエストした場合、どの coval が共有値を更新すべきかについての検討が必要である。

現在の実装では、更新関数を持つ coval のうちで、その coval グループに最後にバインドされたものが更新を行う。coval グループ内で、更新関数を持つものが他にあっても、それらは利用されない。共有値

の更新に利用される関数を持つ coval がアンバインドされた場合、更新関数を持ち、最後にバインドされた別の coval が改めて採用される。

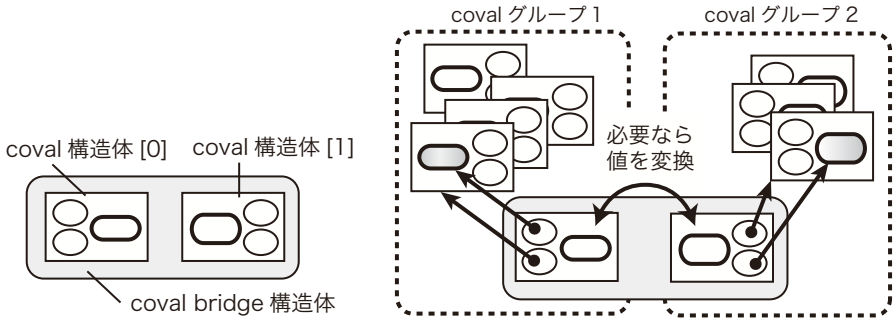
この方法の他に、coval グループ内にある更新関数を順番にすべて呼び出す、などの実装も考えられる。

4 coval bridge

4.1 coval bridge の概要

coval によって共有される値は、同じ型でなければならない。しかし、同じ意味を持ち、共有したい情報が、あるモジュールでは int 型、別のモジュールでは short 型や float 型でプログラムされているという場合も珍しくない。構造体の一部のメンバだけを共有したい場合もある。

また、すでに存在する coval グループと、別の coval グループを新たに連携させたい場合、バインドの方法を変更するのではなく、グループとグループを結合できればモジュール群としての独立性を向上させることができる。



(a) coval bridge 構造体の概念　(b) coval bridge による共有の概念

図 2: coval bridge 構造体と coval グループの結合

このために、coval bridge という構造を導入する coval bridge は図 2(a) のように coval を 2 つ含む構造で、それぞれの coval の共有データの型は異なっていてもよい。図 2(b) のように、2 つの coval は別々の coval グループにバインドし、一方で変更された値を他方に伝えることができる。この際、必要に応じて値を変換して伝えることができる。

ただし、節 1.3 で述べたように、実装のためには coval.c, coval.h の 2 つのファイルのほか、covalbridge.c, covalbridge.h を使い、マクロ `USE_COVAL_BRIDGE` が 1 に定義されている必要がある。

4.2 coval bridge 構造体の宣言と初期化

データ型 T1 と T2 の coval を持つ coval bridge のデータ型は、マクロ `CovalBridgeOf` を使って次のように宣言する。

```
CovalBridgeOf(T1, T2)
```

この構造体は、使用する前に初期化を行う必要がある。このためにマクロ `CovalBridgeInit` を用いる。第 1 、第 2 引数は上記の宣言で用いたのと同じ型引数、第 3 引数に具体的な coval bridge 構造体型へのポインタを指定する。

```
CovalBridgeInit(T1, T2, cov *)
```

例えば、int 型と float 型を持つ coval bridge 構造体の変数 a を初期化するには以下のようにする。

```
typedef CovalBridgeOf(int, float) bridgeIntFloat;
bridgeIntFloat a;

CovalBridgeInit(int, float, &a);
```

参考

マクロ CovalBridgeOf を使って宣言する構造体は、指定されるデータ型によってそれぞれに異なるが、それらの共通部分のデータ型は実装内で bridge_base 型として参照される。ただし、アプリケーション側のプログラミングでは bridge_base 型を使うべきではない。変数や関数の引数には、具体的な coval bridge の型を指定しなければならない（coval 構造体についての注意と同様である）。

4.3 coval bridge のバインドとアンバインド

coval bridge は内部に 2 つの coval 構造体を持ち、これらをそれぞれ別の coval にバインドすることができる。このために、次のマクロ CovalBridge を用いる。なお、brg は何らかの coval bridge 構造体の型、cov は coval 構造体の型とする。第 2 引数は 0 または 1 で、coval bridge 内の coval を示すインデックスである。

```
CovalBridge(brg *, int, cov *);
```

例えば、int 型と float 型を持つ coval bridge 構造体の変数 b を、int 型を共有型とする coval 構造体の変数 a とバインドするには次のようにする。

```
CovalTypeOf(int) a;
CovalBridgeOf(int, float) b;
...
CovalBridgeInit(int, float, &b);
...
CovalBridge(&b, 0, &a);  // 0 は b の構造体のうち、int を持つ coval を指す
```

アンバインドするには、マクロ CovalUnbridge を用いる。第 2 引数は内部の 2 つの coval のどちらをアンバインドするかを示すインデックスで、バインドに用いたものと同様である。

```
CovalUnbridge(brg *, int);
```

構造体を共有値とする coval に coval bridge をバインドする場合、共有値の構造体の特定のメンバだけを指定してバインドすることができる。このために、マクロ CovalBridgeMember が用意されている。

```
CovalBridgeMember(brg *, int, cov *, メンバ名);
```

例を示す。構造体 struct point を共有値とする coval 型の変数 a に対して、そのメンバである x を coval bridge の一方とバインドには以下のようにする。アンバインドにはマクロ CovalUnbridge を用いればよい。

```
struct point {
    float x, y;
};
CovalTypeOf(struct point) a;
CovalTypeOf(float, float) w;
...
CovalBridgeInit(float, float, &w);
...
BridgeCovalMember(&w, 0, &a, x);
```

4.4 値の伝播と型変換

coval bridge は、結びつけた２つの coval グループの間で値を伝える。共有値のデータ型が異なる２つの coval グループをバインドする場合、それらの間で型変換を行う必要がある。整数型同士、あるいは整数と実数のような単純な型変換であれば、coval bridge に型変換を行わせることができる。さらに、数値は符号を逆にしたり、ブール値の真偽を逆にして伝えることもできる。

また、どちらか一方からだけ値を伝えるようにしたり、一方の更新後の値が以前の値と同じ場合には他方に伝えないようにしたりする機能も備えている。

次のマクロ CovalSetBridgeConversion で、値の伝播と型変換の設定が可能である。第１引数 bp は対象となる coval bridge 構造体へのポインタである。第２引数 index はどちら向きの伝播についての指定なのかを表す。index が０ならば、インデックス０の側からインデックス１の側へ、index が１ならばその逆の向きの値の伝播である。第３引数には型変換の有無とその種類を表す定数（下記参照）を指定する。第４引数には、更新後の値が変化しなかった場合に伝播を行うかどうかを指定する真偽値を指定する。真ならチェックを行い、同じ値は伝播しない。偽ならチェックは行わず、つねに伝播をする。

マクロ CovalSetBridgeConversion の返り値は真偽値で、真の場合は問題なく設定できたことを示し、偽の場合は指定に誤りがあったことを示す。

```
CovalSetBridgeConversion(brg *bp, int index, int conv, int check);
```

第３引数として指定する定数を示す。

(1) 変換方法

以下のいずれか１つを指定する。

BR_IntToInt	符号付きの整数型（char, short, int, long）の間の変換を指定する。
BR_IntToFloat	符号付きの整数型から実数型（float, double）への変換を指定する。
BR_UIntToInt	符号なし整数型から符号付き整数型への変換を指定する。
BR_UIntToFloat	符号なし整数型から実数型への変換を指定する。
BR_FloatToInt	実数型から符号付き整数型への変換を指定する。
BR_FloatToFloat	実数型の間の変換を指定する。
BR_NONE	値の更新を伝播しない。この定数は他と組み合わせない。
BR_CopyBytes	値を単なるバイト列としてコピーする。この定数は他と組み合わせない。

(2) 否定・符号反転

上記のうち、BR_NONE と BR_CopyBytes 以外について、以下のいずれかを OR（論理和）で結合して指定できる。

BR_NumNeg	更新された値に対して、その符号を反転した値を伝播する。
BR_LogicNeg	値を論理値とみなし、その否定を伝播する。変換後の型は整数型であること。

4.5 値の伝播と変換関数

節 4.4 で示した型変換以外の変換や、より複雑な処理を行わせたい場合には、値の更新に応じて呼び出される変換用の関数を指定できる。関数の指定にはマクロ CovalSetBridgeFunction を用いる。

```
CovalSetBridgeFunction(brg *bp, int index, 関数ポインタ, int check);
```

第 1 引数は対象とする coval bridge 構造体へのポインタ、第 2 引数は 0 か 1 の整数値で、2 つの coval のうちのどちらを伝播元とする指定なのかを表すインデックスである。第 3 引数が関数へのポインタで、第 4 引数は更新を伝播させるかどうかを制御するための真偽値である（後述）。マクロ CovalSetBridgeFunction は返り値は返さない。

変換関数として指定できる関数は一般には次の形式である。

```
int 関数名 (void *target, const void *src);
```

ここで、第 1 引数が伝播先の coval の共有変数へのポインタ、第 2 引数が伝播元の（更新が発生した）coval の共有変数へのポインタである。実際のプログラミングでは void *型を使わず、具体的なデータ型を指定するべきである。

関数の返り値は真偽値で、マクロ CovalSetBridgeFunction の第 4 引数 check とともに、伝播先の coval グループのコールバック関数を起動させるかどうかを決める役割を担っている。check が偽、または関数の返り値が真の場合、コールバック関数が呼び出される。言い換えると、check が真、かつ関数の返り値が偽の場合、コールバック関数は呼び出されない。コールバック関数が呼び出されない場合も、伝播先の coval の共有変数は値が変更されている可能性があることに注意。

例えば、unsigned char の 0 から 255 までの範囲の数を共有する coval グループと、float 型の 0.0 から 1.0 までの範囲の数を共有する coval グループを互いに連携させるには次のようにすればよい。

```
CovalTypeOf(unsigned char) u;
CovalTypeOf(float) f;
...
/* u と f はそれぞれに coval グループを構成 */

CovalBridgeOf(unsigned char, float) uf_bridge;
CovalBridgeInit(unsigned char, float, &uf_bridge);
CovalBridge(&uf_bridge, 0, &u);
CovalBridge(&uf_bridge, 1, &f);
CovalSetBridgeFunction(&uf_bridge, 0, uch2f, 0);
CovalSetBridgeFunction(&uf_bridge, 1, f2uch, 0);     // (1)
...
int uch2f(float *fp, const unsigned char *up) {
    *fp = (float)*up / 255.0;
}
int f2uch(unsigned char *up, const float *fp) {
```

```
        *up = (int)(*fp * 255.0) & 0xff;
    }
```

なお、上記の例で、float 側の値の変化を unsigned char 側に伝えないようにしたい場合は、(1) の部分を次の (2) のようにすればよい。

```
CovalSetBridgeConversion(&uf_bridge, 1, BR_NONE, 0); // (2)
```

5 著作権および諸注意

本稿で説明した coval 0.81 のソースコードは下記サイトからダウンロード可能である。

http://www.cc.kyoto-su.ac.jp/~ogihara/coval

このソースコードは自由に利用できるが、このソフトウェアにはいかなる保証もない。ソフトウェアの使用によって何らかの損害を被ったとしても、作者はその責を一切負わない。本稿に関しても、記述の正しさに保証はないことに注意されたい。また、coval のソースコードおよび関係文書の著作権は放棄されていない。coval の将来的な保守や機能拡充は保証されたものでなく、仕様および実装は予告なく変更されることがある。

参考文献

[1] 荻原剛志: 手続き型言語におけるデータバインディング機構の提案と構造化設計への適用, 情報処理学会論文誌, Vol.54, No.4 (2013).

[2] 荻原剛志: 共有変数を用いたバインド方式の提案とソフトウェア開発への応用について, 情報処理学会ソフトウェア工学研究会報告, SE-171(23), pp.1–8 (2011).

索 引