連載

SUPER COLLIDER チュートリアル (5)
SUPER COLLIDER TUTORIALS (5)

美山 千香士
Chikashi Miyama
ケルン音楽舞踏大学
Hochschule für Musik und Tanz Köln

概要

本連載では、リアルタイム音響合成環境の SuperCollider(SC) の使い方を、同ソフトを作品創作や研究のために利用しようと考えている音楽家、メディア・アーティストを対象にチュートリアル形式で紹介する。
SuperCollider(SC) is a realtime programming environment for audio synthesis. This article introduces SC to musicians and media artists who are planning to utilize the software for their artistic creations and researches.

1. 今回の目標：バスを理解する

当連載ではこれまで SC によるシンセサイザーやエフェクトの作り方、MIDI 機器との連携、メロディーの演奏の仕方などを紹介してきた。これらを組み合わせてより大規模なプログラムを制作する時に要となるのが**バス** (Bus) という機能である。今回はこのバスについて、例を挙げつつ紹介していく。

2. バスを使う意味

本連載の第 2 回で学習したように、SC では様々な Ugen を組み合わせた一連のオーディオ処理プロセスを **SynthDef** で定義する事で、自分だけの「楽器」を作る事ができる。例えば以下のリスト 1 では Saw で生成したノコギリ波にエンベロープを施し、さらにローパス・フィルタとリバーブをかける一連の処理を SynthDef を用いて *combined* という名前で定義し、その後、定義した SynthDef を **Synth** を用いて実行している。

リスト 1. Synth の例

```
1 SynthDef("combined",{
2   arg freq=440, amp=0.5;
3   e=Env([0,1,1,0],[0.1,0.5,0.4]);
4   g=EnvGen.ar(e,doneAction:2);
5   a=Saw.ar(freq,amp*g);
6   l=LPF.ar(a,700);
7   r=FreeVerb.ar(l,0.5,0.8);
8   Out.ar(0,[r,r]);
9 }).load;
10
11 Synth("combined");
```

しかし、このプログラムには 2 つの問題がある。1 つ目の問題は、doneAction:2 が **EnvGen** に与えられているため、エンベロープが終了した時点（開始から 1 秒後）でこの Synth は解放され、FreeVerb の付加した残響音が不自然に途中で切れてしまい、最後まで聞こえない事。

リスト 2. Synth を用いたアルペジオの演奏

```
1 Routine({
2   [60,64,67].do{
3    arg pt;
4    Synth("combined",["freq",pt.midicps]);
5    0.1.wait;
6   }
7 }).play
```

2 つ目は、この SynthDef を用いて、例えばリスト 2 のようにド・ミ・ソのアルペジオを、第 2 回のチュートリアルで勉強した **Routine** を使って演奏した場合、Saw や EnvGen のように個々の音の波形を生成しエンベロープを施す Ugen だけでなく、ローパス・フィルターとリバーブのように音を加工する Ugen もこの SynthDef の定義に含まれているため、図 1 のように各 Synth に個別のローパス・フィルター、リバーブが生成されてしまい、無駄な CPU 負荷が生じる事である。

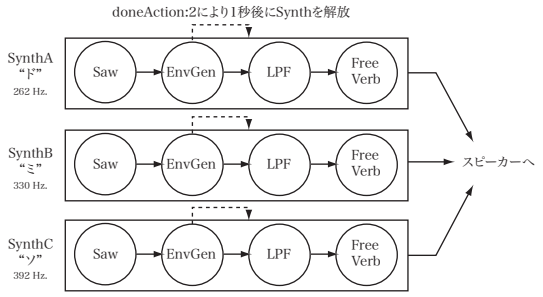

図 1. SynthDef *combined* の問題点

ここで、もし、図 2 の示すように、1 つ 1 つの音を作り出す「生成用」の SynthDef と、生成した音を加工する「加工用」の SynthDef を個別に定義し、複数の「生成用」の Synth が出力したオーディオ信号を単一の「加工用」の Synth に送信し、「加工用」の Synth で全ての「生成用」の Synth の音をまとめて加工することができれば、より効率のよい処理が可能となるはずである。しかし、これを実現するためには、ある Synth から別の Synth に音を「送る」メカニズムが必要となる。そのメカニズムこそが今回のテーマである**バス**である。

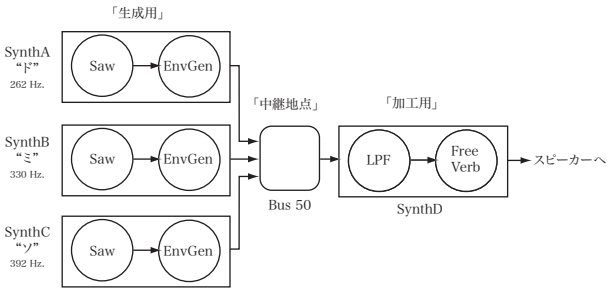

図 2.「生成部」と「加工部」の分離

3. バスとは何か？

バスとは DAW やデジタルミキサーなどでよく使われる用語で、一般的には「信号経路」という意味であるが、SC でのバスは簡単に言えばオーディオ信号の**中継地点**を意味する。これを用いることにより、例えばある Synth で生成した信号を一度バス（＝中継地点）に送り、その後、他の Synth がその中継地点から音を受け取り、音を加工し、スピーカーに送るというような事が可能となる。バスは非常に柔軟であり、1 つのバスに複数の Synth から同時に音を送る事も、複数の Synth が 1 つのバスから同時にオーディオ信号を取得することも可能である。

リスト 3 はリスト 1 をバスを利用して書きなおしたものである。先ほど 1 つの SynthDef、*combined* として定義していたものを、バスを用いると、「生成用」の Synth、*generate* と「加工用」の Synth、*process* に分割して定義することができる。

リスト 3. バスの例

```
1  SynthDef("generate",{
2    arg freq=440,amp=0.5;
3    e=Env([0,1,1,0],[0.1,0.5,0.4]);
4    g=EnvGen.ar(e,doneAction:2);
5    a=Saw.ar(freq,amp*g);
6    Out.ar(50,a);
7  }).load;
8
9  SynthDef("process",{
10   i=In.ar(50);
11   l=LPF.ar(i,700);
12   r=FreeVerb.ar(l,0.5,0.8);
13   Out.ar(0,[r,r]);
14 }).load;
```

リスト 3 から分かるように、バスにオーディオ信号を送るには、**Out** を、バスからオーディオ信号を取得するには **In** を用いる。SC ではバスを複数同時に使用する事ができるため、個々のバスには番号が与えられている。例えば、SynthDef、*generate* では、ノコギリ波の信号を Out.ar(50, a) でバスの 50 番に送っている。そして SynthDef、*process* では 50 番に送られた信号を In.ar(50) を用いて取得し、それをローパス・フィルターとリバーブで加工している。Out.ar はスピーカーにオーディオ信号を送る Ugen として、既にチュートリアルの第 2 回で一度利用しているが、ここではスピーカーにではなくバスにオーディオ信号送るために使用している。この Out のバスとオーディオ入出力との兼ね合いについては後述する。

バスを用いて 2 つ以上の Synth を連携させる時の注意点として、必ずオーディオ信号を「受け取る側」そして「送る側」の順番で実行しなくてはならない事があげられる。例えば、リスト 3 の例は、*process* が信号を受け取る側、*generate* が送る側であるので、リスト 4 のように *process*、*generate* の順番で実行する。これをもし逆の順番で実行した場合には音が聞こえてこないだけでなく、エラーも出力されない。故に、バスを使う場合に Synth を実行する順番に細心の注意を払う必要がある。これは SC サーバー内での「ノード」と呼ばれる構造に関連した事項であるが、詳細は次回以降に扱う。

リスト 4. Synth 実行の順番

```
1 Synth("process");
2 Synth("generate");
```

In

指定したバスからの信号を取得する。.ar でオーディオ信号、.kr でコントロール信号を得る。

.ar(*bus*, *numChannels*)

.kr(*bus*, *numChannels*)

bus･･･ 取得するバスの番号。複数取得する場合は最初の番号。

numChannels･･･ 取得するチャンネルの数。

リスト 5 はリスト 2 のアルペジオの演奏をバスを用いて行ったものである。

リスト 5. バスを用いたアルペジオの演奏

```
1 Synth("process");
2 Routine{
3   [60,64,67].do{
4     arg pt;
5     Synth("generate",["freq",pt.midicps]);
6     0.1.wait;
7   }
```

```
8 }.play
```

リスト 2 とリスト 5 をそれぞれ実行して、ステータス・バーで使われている Ugen の数や CPU の使用率を比較してみると、図 3 の示すように、明らかに分割した方が使用している Ugen の数も少なく、また CPU 使用率も低い事がわかる。このように、SynthDef をバスを用いて「生成部」と「加工部」に分割する事で、余計な CPU 負荷を生じさせる事なく音の生成と加工が可能となる。無論このような小規模なプログラムの場合は CPU に与える負荷の違いは大きなものではないが、例えば、より複雑なエフェクトを施す SynthDef を定義し、それを用いて 100 音を同時に生成するといった場合には、大きな差が生まれる。

図 3. リスト 2(上) と 5(下) のパフォーマンス比較

また、リスト 5 では、doneAction:2 により FreeVerb がエンベロープの終了とともに解放されないため、残響効果を最後まで聞き取ることができる。

4. バスの信号を可視化する

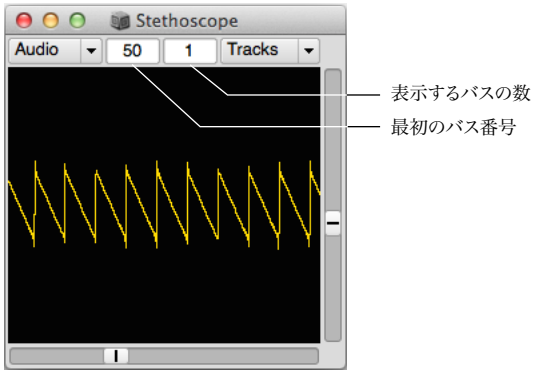

図 4. scope によるバスの可視化

バスを複数同時に使用するようになると、Synth 間の連結が複雑になり、指定したバスに任意の信号が送られているかを確認する必要が生じる。これを行うには、リスト 6 のようにバスにオーディオ信号を送る Synth を実行し、その後、SC サーバーに.scope メッセージを送る。すると、図 4 のようにバスの信号をスコープに表示させる事ができる。この時.scope メッセージの第 1 引数は表示するバスの数、第 2 引数は表示を開始するバスの番号となる。故に、.scope(1,50) を送ると 50 番のバス 1 チャンネルが表示される。また表示されるスコープ・ウインドウの GUI を操作する事によって、表示するバスとその数を任意に変更する事も可能である。

リスト 6. バスをスコープで確認

```
1 Synth("generate");
2 s.scope(1,50);
```

5. バスとオーディオ・チャンネルの関係

リスト 3 では 2 つの Synth、*generate* と *process* を繋ぐために 50 番のバスを利用した。このバス番号は、ユーザーが基本的に任意に決めてよいが、幾つかの点に留意する必要がある。

まず、デフォルトでは、バスは 128 個、つまり 0 から 127 までしか使用できない。もし、それ以上のバスを利用したい場合は、サーバーを起動する前にリスト 7 の要領でバスの数を変更する必要がある。

リスト 7. オーディオ・バスの数の変更

```
1 s.options.numAudioBusChannels=256;
```

また 3 や 7 など低いバス番号を使う時も注意が必要である。それには以下のような理由がある。

SC サーバーは使用しているハードウェアの構成、つまりコンピュータのマイク・ライン入力や HDMI 入出力などの数に関わらず、デフォルトで 8 チャンネルのオーディオ入力と 8 チャンネルのオーディオ出力、計 16 チャンネルのオーディオ入出力を確保する。現在、確保されているオーディオ入出力チャンネルの数を確認するには、リスト 8 のように、SC サーバーにメッセージを送る。すると、設定を変更していない限り、現在確保されている入出力オーディオ・チャンネル数である「8」が 2 度ポスト・ウインドウに表示される。

リスト 8. 入出力チャンネルの確認

```
1 s.options.numOutputBusChannels.postln;
2 s.options.numInputBusChannels.postln;
```

しかし、例えば筆者の環境ではオーディオ入力は 2 チャンネル、出力も 2 チャンネルしかないため、デフォルトのままでは余剰な入出力チャンネルが確保されている事になる。無論、これまでのように、この設定のままで SC サーバーを使用しても問題はないが、ハードウェアの入出力チャンネル数に合わせて、SC の確保する入出力チャンネル数を、リスト 9 のように変更する事も可能である。

リスト 9. 入出力チャンネルの変更

```
1 s.options.numOutputBusChannels=2;
2 s.options.numInputBusChannels=2;
```

リスト 9 では入出力のチャンネル数をそれぞれ 2 チャンネルに設定し直している。これにより SC サーバーは起動時に入力・出力ともに 2 チャンネル、合計 4 チャンネルのオーディオ入出力チャンネルを確保する。このサーバーを起動する前に確保したオーディオ入出力チャンネル数はバスと密接な関係があり、SC サーバーを起動した時に、N 個のオーディオ出力と M 個のオーディオ入力が確保されている場合は、バスの「0」から「N-1」が自動的にオーディオ出力に送られ、「N」から「M-1」が自動的にオーディオ入力からの信号を得る。

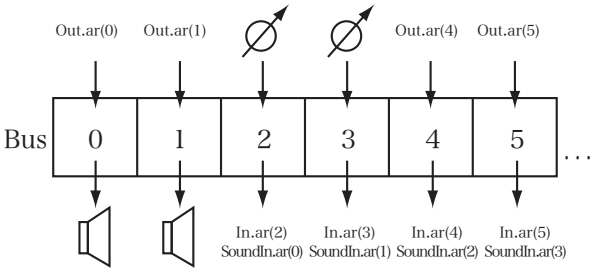

図 5. リスト 8 実行後のオーディオ入出力とバスの関係

例えば、リスト 9 を実行してオーディオ入出力チャンネル数をそれぞれ 2 に設定すると、図 5 のようにバスの 0 と 1 チャンネルはオーディオ出力に送られ、2 から 3 チャンネルはオーディオ入力からの信号を受けとる事となる。故に、リスト 9 を実行し、サーバーを立ち上げた後に、リスト 10 のようなコードを実行すると、入力信号が出力にバイパスされる（直接送られる）こととなる[1]。

リスト 10. オーディオ入出力のバイパス

```
1 SynthDef("bypass",{
2   Out.ar(0,In.ar(2, 2));
3 }).load;
4
5 Synth("bypass");
```

このように、低い番号を持つバスは、ハードウェアのオーディオ入出力と接続している可能性があるため、現在何チャンネルのオーディオ入出力が SC サーバーに確保されているのかを念頭におき、Synth 間でのオーディオ信号の受け渡しに使うバスとしてこれらを使用しないように留意する必要がある。

コンピュータにオーディオ・インターフェースなどの機器を繋ぎ、より多くのオーディオ入出力を使う時は、リスト 9 の要領で、入出力チャンネル数を増やす必要がある。このように、ハードウェアの増設などによって、入出力チャンネル数の設定を変更しなければならない時、例えば、オーディオ出力数を 32 チャンネルに設定

[1] 注：ハウリングを防止するため、ラップトップでこのプログラムを実行する際はヘッドフォンを着用の事。

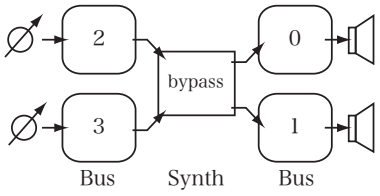

図 6. リスト 10 のバスと Synth の関係

した場合には、オーディオ入力をとるために、In の第 1 引数を 32 に変更しなくてはならない。このように In を用いてマイクやラインからのオーディオ入力を得ていると、ハードウェア構成を変更した際にプログラムを変更しなくてはならないため、オーディオ入力を得るには前回のチュートリアルで紹介した **SoundIn** が推奨される。SoundIn は In とほぼ同じ機能を提供する Ugen であるが、In と違いチャンネルに 0 番を指定すると、オーディオ出力のチャンネル数の設定に左右されることなく、最初のオーディオ入力が接続されているバスからのオーディオ信号を得る事ができる。

6. 2 種類のバス

.kr と.ar については第 2 回のチュートリアルで学習した。.kr はコントロール・レート、.ar はオーディオ・レートの意味で、コントロール・レートで作成した信号は直接スピーカーに送って聞くことは不可能だが、他の Ugen のパラメータに代入し、そのパラメータを動的に変化させる事ができる。例えば、リスト 11 では SinOsc で作られたコントロール・レートの波形 (LFO) が 2 つ目の SinOsc の周波数をコントロールすることで、ビブラートのような効果を得ている。

リスト 11. コントロール信号の例

```
1 SynthDef("vibrato",{
2   l=SinOsc.kr(5,0,100);
3   x=Saw.ar(l+440,0.5);
4   Out.ar(0,[x,x])
5 }).load;
6
7 Synth("vibrato");
```

SC のバスにはオーディオ信号を送るオーディオ・バス以外にもこのようなコントロール信号を送ることのできるコントロール・バスがある。コントロール信号をバスに送るためには、Out.ar の代わりに Out.kr を用いる。リスト 12 はリスト 11 のプログラムをコントロール・バスを用いて 2 つの Synth に分割して記述したものである。

リスト 12. コントロール・バスの例

```
1 SynthDef("lfo",{
2   l=SinOsc.kr(5,0,100);
```

```
  Out.kr(0,l);
}).load;

SynthDef("saw",{
  arg freq;
  l=In.kr(0);
  x=Saw.ar(l+freq,0.5);
  Out.ar(0,[x,x]);
}).load;
```

この SynthDef をリスト 13 の要領で実行すると、Synth、*lfo* で、振幅が 100 で周波数が 5 Hz. の正弦波が作られ、その信号がコントロール・バスの 0 番に送られる。そして、SynthDef、*saw* は 0 番のコントロール・バスから信号を取り出し、ノコギリ波を生成する Saw.ar の周波数にその信号と引数 freq を加算したものを適用し、リスト 11 と同様のビブラートのかかったノコギリ波を生成する事ができる。

リスト 13. リストの実行

```
Synth("saw",["freq",440]);
Synth("lfo");
```

前述したように、複数の Synth が同一のバスから信号を得ることも可能である。これを利用して例えば、リスト 14 のように複数の Synth を同じバスに接続し、2 つの Synth のパラメータを単一の LFO を同期させるなどという利用が考えられる。

リスト 14. LFO の同期

```
Synth("saw",["freq",440]);
Synth("saw",["freq",540]);
Synth("lfo");
```

このリストの 2 つ *saw* はどちらもコントロール・バスの 0 番を参照しているため、2 つの Synth によって奏される音の周波数は異なるが、ビブラートが同期している。

コントロール信号を扱うコントロール・バスはオーディオ信号を扱うオーディオ・バスと完全に分離しており、相互に干渉する事はない。また、オーディオ・バスと異なり、コントロール・バスはデフォルトで 4096 チャンネルまで使う事ができる。

7. バス番号を引数を用いて指定する

今までの例では、Out、In に与える引数は全て、50 や 0 などの定数であった。しかし、実際のプログラミングでは、より SynthDef を柔軟に活用するために、Out 及び、In のバス番号を SynthDef 内で引数 (arg) を介して指定するのが一般的である。例えばリスト 15 はリスト 12 に変更を加え、Out と In のバスをそれぞれ引数で指定できるようにしたものである。

リスト 15. バス番号を引数で指定

```
SynthDef("lfo",{
  arg outBus;
  l=SinOsc.kr(5,0,100);
  Out.kr(outBus,l);
}).load;

SynthDef("saw",{
  arg inBus,freq;
  l=In.kr(inBus);
  x=Saw.ar(l+freq,0.5);
  Out.ar(0,[x,x]);
}).load;
```

SynthDef をこのように定義した場合、リスト 16 のように、コントロール信号の送信先、または受信元のバスを実行時に決定する事ができるため、定義した SynthDef をより柔軟に用いる事ができる。

リスト 16. 引数を介した実行時のバス番号の指定

```
Synth("saw",["inBus",50,"freq",440]);
Synth("lfo",["outBus",50]);
```

8. バスの接続をリアルタイムに変更する

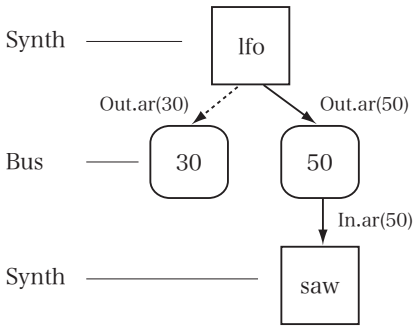

図 7. 実行時のバス接続変更

バスの接続は Synth を実行した後も任意に変更することが可能である。リスト 17 ではリスト 16 と同様に LFO を 50 番のバスに送り、*saw* がそれを受け取って、ビブラートのかかったノコギリ波が生成される。リスト 16 との唯一の違いは最初の Synth、*lfo* が変数 x に代入されている点である。

リスト 17. Synth「lfo」を x に代入

```
Synth("saw",["inBus",50,"freq",440]);
x=Synth("lfo",["outBus",50]);
```

その後、x に set メッセージを送ることにより、outBus を他の番号に変更すると、2 つの Synth 間のバスによる接続が中断され、ビブラートが停止する (リスト 18)。

リスト 18. outBus の変更

```
x.set("outBus",30);
```

しかし、もう一度リスト 19 のように再度バスを 50 番変更した場合、両 Synth が再接続され再びノコギリ波にビブラートがかかる。

リスト 19. outBus を再び戻す

```
1 x.set("outBus",50);
```

このように、SC ではバスの接続はいつでも自由に変更する事ができる。これを応用すれば、例えば、ライブパフォーマンスの際に、バスの接続を任意に変更し、音を生成する Synth のパラメータを様々な LFO でコントロールしたり、素材音に次々に多様なエフェクトを施していく事も可能である。

9. まとめ

今回は、SC のバス機能について学習した。バスによる Synth の連携は便利であるが、あらゆる側面で有効な手段とはいえない。バスの使用により SynthDef を分割しすぎると、Synth の実行の順番の問題や、どのようにバスを繋ぐかを事前に入念に設計する手間がかかることとなる。故に、バスをプログラムに利用する際は、その利用によって、CPU 負荷が大幅に軽減されるなどの明白なメリットがあるかどうかを事前に十分に検討する必要がある。次回は、このバス機能に密接に関連した Synth の計算順序に関わる「ノード」という概念について学習する。

10. 参考文献

[1] *SuperCollider*, http://supercollider.sourceforge.net(アクセス日 2014 年 9 月 29 日)

[2] Wilson, S., Cottle, D. and Collins, N. (eds), *The SuperCollider Book* The MIT Press, 2011

11. 著者プロフィール

11.1. 美山 千香士 (Chikashi Miyama)

作曲家、電子楽器創作家、映像作家、パフォーマー。国立音楽大学音楽デザイン学科より学士・修士を、スイス・バーゼル音楽アカデミーよりナッハ・ディプロムを、アメリカ・ニューヨーク州立バッファロー大学から博士号を取得。作曲・コンピュータ音楽を莱孝之、エリック・オニャ、コート・リッピ氏らに師事。Prix Destellos 特別賞、ASCAP/SEAMUS 委嘱コンクール 2 位、ニューヨーク州立大学学府総長賞、国際コンピュータ音楽協会賞を受賞。2004 年より作品と論文が国際コンピュータ音楽会議に 15 回入選、現在までに世界 19 カ国で作品発表を行っている。 2011 年、DAAD(ドイツ学術交流会) から研究奨学金を授与され、ドイツ・カールスルーエの ZKM で客員芸術家として創作活動に従事。近著に「Pure Data-チュートリアル＆リファレンス」(Works Corporation 社) がある。現在ドイツ・ケルン音楽大学講師。スイス・チューリッヒ芸術大学コンピュータ音楽・音響研究所 (ICST) 研究員。バーゼル造形大学のプロジェクト「Experimental Data Aesthetics」プログラマ。2013 年に松村誠一郎氏と日本語の Pure Data ポータル Pure Data Japan(http://puredatajapan.info) を創設。公式ウェブサイト：http://chikashi.net