MỤC LỤC

Chương 12. Ứng dụng web audio phần 2.....5
12.1 Vị trí của chương trong cuốn sách.....6
12.2 Lập lịch tham số — đổi núm mà không kêu tách.....8
12.3 Rack hiệu ứng — chuỗi khối bật tắt được.....13
12.4 Trộn động — khi bản trộn tự đổi theo sự kiện.....17
12.5 Tối ưu tài nguyên — làm ít đi chứ không làm nhanh hơn.....21
12.6 Vòng đời và bộ nhớ — node đã chết đi đâu.....27
12.7 Đóng gói ứng dụng — tách máy âm thanh khỏi giao diện.....30
12.8 Kiểm và đo — làm sao biết lần sửa vừa rồi không làm sai gì.....33
12.9 Thực hành, bài tập và tổng kết.....37

DANH MỤC HÌNH VẼ

Hình 12.1 Bảy mục kỹ thuật của chương 12.....6
Hình 12.2 Bốn khoảng cách giữa một đồ thị chạy được và một ứng dụng giao được.....6
Hình 12.3 Nhiễu răng cưa của phép gán thẳng, và cách sửa bằng một dòng.....9
Hình 12.4 Mỗi loại tham số một cách đổi và một hằng số thời gian.....10
Hình 12.5 Quên huỷ lịch cũ, và cách huỷ rồi neo lại cho đúng.....11
Hình 12.6 Vì sao mọi lệnh lập lịch đều cộng thêm một khoảng an toàn.....12
Hình 12.7 Kiến trúc một rack: khe, nhánh bypass, và đường khô song song.....13
Hình 12.8 Hai cách bypass, và cái giá tính bằng CPU của mỗi cách.....14
Hình 12.9 Hai luật trộn khô-ướt, và mức đo được trên toàn dải núm.....16
Hình 12.10 Đổi thứ tự hai khối trong rack, và hai con số đổi theo.....17
Hình 12.11 Ba tầng của một bản trộn: nguồn, nhóm, và bus gửi.....18
Hình 12.12 Đường điều khiển ducking, và lời nổi lên bao nhiêu trên tiếng ra.....19
Hình 12.13 Sự kiện nào đổi tham số nào, đổi ra sao, và điểm dễ sai đi kèm.....21
Hình 12.14 Cùng một tiếng, hai cách mắc bộ lọc, và tỉ số giữa hai cách.....22
Hình 12.15 Cái giá của việc mỗi giọng mang một phòng vang riêng.....24
Hình 12.16 Trần đa âm: cái được tính bằng CPU và cái mất tính bằng decibel.....25
Hình 12.17 Sáu việc tối ưu xếp theo tỉ số được trên công bỏ ra.....26
Hình 12.18 Sáu bước của một giọng, và bước duy nhất người ta hay bỏ sót.....27
Hình 12.19 Cái giá của node bị bỏ sót, đo bằng CPU và bằng bộ nhớ.....28
Hình 12.20 Cái gì phải dọn, cái gì phải giữ, và dọn bằng cách nào.....30
Hình 12.21 Ba mô-đun, và giao diện chỉ được gọi qua một ranh giới duy nhất.....30
Hình 12.22 Năm mốc từ lúc trang mở tới lúc ứng dụng sẵn sàng phát.....31
Hình 12.23 Hai mảnh mã: thứ dựng một lần, và thứ sinh ra hàng nghìn lần.....32
Hình 12.24 Dựng hai lần cùng một đồ thị, và sai lệch giữa hai lần.....33
Hình 12.25 Tám dòng phải qua trước khi giao một ứng dụng Web Audio.....36
Hình 12.26 Chương 12 gói lại trong sáu con số.....38

DANH MỤC BẢNG BIỂU

Bảng 12.1 Bảy mục và câu hỏi mà mỗi mục trả lời bằng số.....7
Bảng 12.2 Ba nhịp thời gian cùng chạy trong một ứng dụng.....8
Bảng 12.3 Ba cách đưa tham số tới giá trị mới.....10
Bảng 12.4 Khoảng an toàn nên dùng trong bốn tình huống.....12
Bảng 12.5 Ba thứ một rack có mà một chuỗi connect thường không có.....13
Bảng 12.6 Cùng hai khối, đổi thứ tự, và hai con số đổi hẳn.....16
Bảng 12.7 Bộ nhóm tối thiểu cho một ứng dụng trò chơi hoặc phát thanh.....18
Bảng 12.8 Ba cách sinh tín hiệu điều khiển cho ducking.....20
Bảng 12.9 Khối nào gộp chung được, khối nào không.....23
Bảng 12.10 Vang riêng và vang chung — thời gian dựng đo được.....24
Bảng 12.11 Sáu bước, ai gọi, và chuyện gì sai nếu bỏ.....27
Bảng 12.12 Bỏ lại K chuỗi node đã phát xong — thời gian dựng đo được.....28
Bảng 12.13 Chi phí tạo và nối node, đo trên bốn cỡ đồ thị.....29
Bảng 12.14 Ba mô-đun: việc của mỗi bên, và thứ mỗi bên bị cấm.....31
Bảng 12.15 Ba mức kiểm và khi nào dùng mức nào.....34
Bảng 12.16 Ba số đo trong lúc chạy, ngưỡng, và việc phải làm khi vượt.....35
Bảng 12.17 Kiểm mỗi dòng của bảng bằng cách nào, và con số để so.....36

CHƯƠNG 12
ỨNG DỤNG WEB AUDIO PHẦN 2

Mục tiêu của chương

Sau chương này, người đọc phải làm được những việc sau.

- Viết vào một AudioParam bằng ba cách chuẩn, chọn đúng cách cho từng loại tham số, và *đo* được mức nhiễu răng cưa đã giảm.
- Huỷ và neo lại lịch cũ đúng cách khi người dùng đổi ý giữa chừng, và giải thích được vì sao thiếu dòng neo thì sai.
- Dựng được một rack hiệu ứng có bypass thật, và chứng minh bằng số rằng đặt gain về 0 không tiết kiệm gì.
- Chọn đúng luật trộn khô-ướt và đo được độ gợn mức của luật do người đọc chọn trên toàn dải núm.
- Giải thích được ảnh hưởng của thứ tự khe trong rack bằng hai chỉ số đỉnh và độ to.
- Dựng được kiến trúc bus và nhóm, cài được ducking hai chiều, và đo được phần lời nổi lên trên tiếng ra.
- Chỉ ra được khối nào gộp chung được và khối nào không, và đo được tỉ số tiết kiệm của mỗi cách gộp.
- Đặt được trần đa âm có mờ ra, và nói được cái giá phải trả cho nó.
- Viết đúng vòng đời sáu bước của một giọng, và phát hiện được node rò bằng cách đếm chứ không bằng công cụ đo bộ nhớ.
- Tách được máy âm thanh khỏi giao diện, và dựng được một phép kiểm hồi quy chạy trong vòng kiểm tự động.

Chương này trả lời bằng số những câu hay được trả lời bằng cảm tính

Đặt gain về 0 có tiết kiệm máy không? Đo được: 519 so với 517 mili giây, tức không tiết kiệm gì. Chỉ ngắt nối mới rẻ, và rẻ 3,5 lần.

Một bus vang chung rẻ hơn bao nhiêu so với mỗi giọng một phòng? Đo được: 3,2 lần ở bốn giọng, và 16,2 lần ở ba mươi hai giọng.

Node đã phát xong mà bỏ sót việc ngắt thì tốn gì? Đo được: chậm hơn 1,77 lần — và công cụ đo bộ nhớ của trình duyệt *không* nhìn thấy nó.

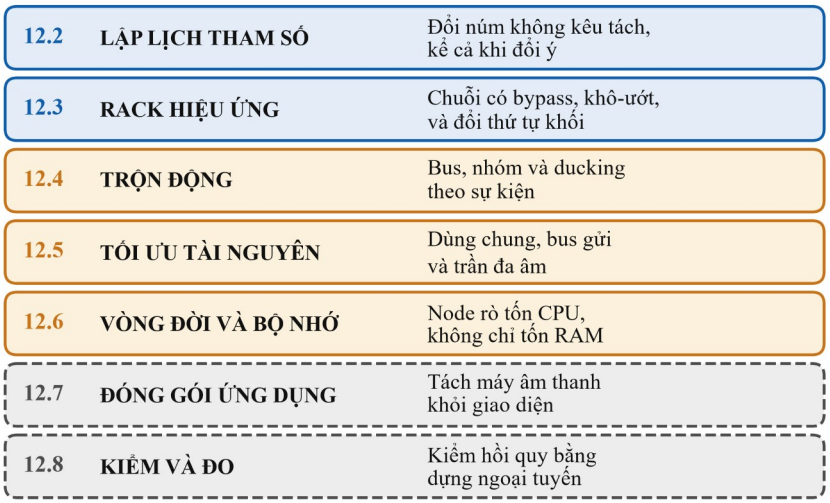
Toàn bộ mã JavaScript trong bài đã chạy thật trên một bản cài đặt Web Audio đầy đủ, kết quả đổ ra tệp rồi đo lại bằng mô-đun Python của các chương trước.

12.1 Vị trí của chương trong cuốn sách

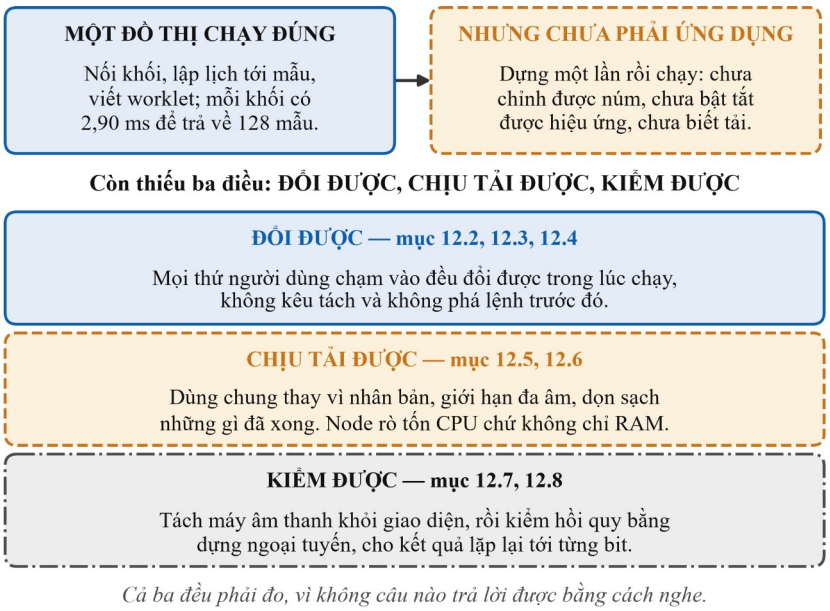
12.1.1 Chương 11 dựng được đồ thị. Chương này dựng được ứng dụng

Cuối chương 11, người đọc đã dựng được một đồ thị node chạy thật: có nguồn phát, có bộ lọc, có worklet riêng, và có lịch sự kiện chính xác tới một mẫu. Nhưng một đồ thị chạy được chưa phải là một ứng dụng.

Khoảng cách giữa hai thứ ấy không nằm ở phép xử lý. Nó nằm ở những câu hỏi mà chương 11 chưa cần trả lời: khi người dùng kéo một thanh trượt thì viết vào tham số như thế nào cho khỏi kêu tách; khi tắt một hiệu ứng thì máy có rẻ đi thật không; khi có ba mươi giọng cùng lúc thì cái gì nên dùng chung; và khi một tiếng đã tắt thì những node của nó đi đâu.



Hình 12.1 Bảy mục kỹ thuật của chương 12



Hình 12.2 Bốn khoảng cách giữa một đồ thị chạy được và một ứng dụng giao được

12.1.2 Mọi câu trả lời trong chương này đều là số đo

Chương 11 đã đặt ra một nguyên tắc: mã JavaScript in trong bài là mã đã chạy thật, và mọi con số là số đo trên tiếng ra. Chương 12 giữ nguyên nguyên tắc ấy, và dùng nó cho một loại câu hỏi khác — những câu hỏi mà người ta hay trả lời bằng cảm tính.

Ví dụ câu hỏi quen thuộc nhất: đặt gain của một nhánh về 0 có tiết kiệm máy không? Ai cũng có một ý kiến. Mục 12.3.3 đo và cho ra con số 519 mili giây so với 517 — nghĩa là *không* tiết kiệm gì hết. Chỉ ngắt nối mới rẻ, và rẻ 3,5 lần.

Bảng 12.1 Bảy mục và câu hỏi mà mỗi mục trả lời bằng số

Mục	Câu hỏi	Trả lời đo được
12.2	Viết vào tham số thế nào cho khỏi kêu tách?	Ramp thay vì gán thẳng: sạch hơn 71 decibel
12.3	Bypass bằng gain = 0 có rẻ không?	Không rẻ chút nào; ngắt nối rẻ 3,5 lần
12.3	Luật wet/dry nào giữ được mức?	Tuyến tính hõm 2,99 dB; đẳng công suất gợn 0,03
12.4	Ducking tự động nâng lời lên bao nhiêu?	Từ dưới nhạc 2,62 dB lên trên nhạc 8,43 dB
12.5	Một bus vang chung rẻ hơn bao nhiêu?	Từ 3,2 lần ở 4 giọng lên 16,2 lần ở 32 giọng
12.6	Node đã phát xong mà không ngắt thì tốn gì?	Chậm hơn 1,77 lần, và bộ nhớ thật không trả lại
12.8	Dựng lại một cảnh có ra đúng kết quả cũ không?	Đúng từng bit — sai lệch bằng 0 trên 88.200 mẫu

12.1.3 Một nhận xét về thứ tự các mục

Bảy mục của chương này xếp theo một trật tự có chủ ý: từ thứ nghe được sang thứ đo được, rồi mới tới thứ giao được.

- **Hai mục đầu — 12.2 và 12.3 — là chuyện tai nghe thấy.** Tiếng tách, tiếng bụp khi bật tắt, mức tụt khi kéo núm wet/dry. Người dùng phàn nàn về đúng những thứ này.
- **Ba mục giữa — 12.4 tới 12.6 — là chuyện máy chịu được.** Trộn động, dùng chung tài nguyên, và dọn dẹp. Người dùng không nghe thấy chúng, nhưng khi chúng sai thì máy nóng lên và tiếng bắt đầu ngắt quãng.
- **Hai mục cuối — 12.7 và 12.8 — là chuyện giao được.** Tách mô-đun và kiểm hồi quy. Đây là phần biến một bài tập thành một sản phẩm mà người khác dám sửa.

Một lời dặn trước khi vào bài

Chương này có rất nhiều con số, và cám dỗ lớn nhất là học thuộc chúng. Đừng. Con số của máy dựng cuốn sách này không phải con số của máy người đọc, và càng không phải con số của máy người dùng.

Thứ đáng học là *cách đo* và *hình dạng* của kết quả: cái nào tăng tuyến tính theo số giọng, cái nào gần như nằm ngang, cái nào không đổi gì cả. Hình dạng ấy thì đúng trên mọi máy.

Mỗi phép đo trong bài đều kèm đủ thông tin để chạy lại. Bài tập cuối chương yêu cầu chạy lại ít nhất ba phép trên máy của người đọc.

12.2 Lập lịch tham số — đổi núm mà không kêu tách

12.2.1 Lỗi phổ biến nhất: gán thẳng vào value mỗi khung hình

Nối một thanh trượt HTML với một AudioParam là việc đầu tiên ai cũng làm, và hầu như ai cũng làm sai theo cùng một kiểu. Mã sai trông rất hợp lý.

Liệt kê 12.1 Cách viết ai cũng nghĩ tới đầu tiên — và nó sai

```
slider.addEventListener("input", (e) => {
  gainNode.gain.value = Number(e.target.value);   // SAI
});
```

Vấn đề không nằm ở chỗ giá trị sai. Giá trị hoàn toàn đúng. Vấn đề nằm ở chỗ trình duyệt chỉ sinh sự kiện input mỗi khung hình một lần — sáu mươi lần mỗi giây, tức cách nhau 16,7 mili giây — trong khi Web Audio chạy 344,53 khối mỗi giây. Giữa hai lần gán, tham số đứng nguyên một chỗ rồi nhảy một bước.

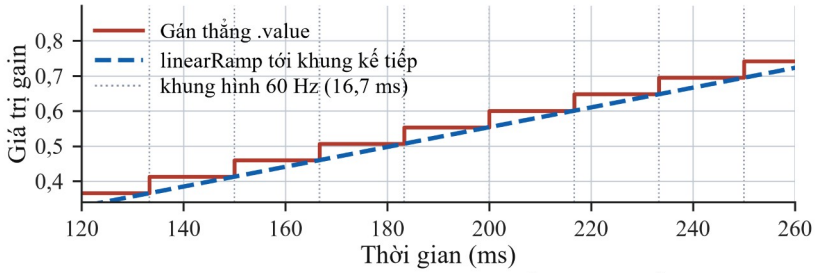
$$\Delta v_k = \left| v(t_{k+1}) - v(t_k) \right|, \qquad t_{k+1} - t_k = \frac{1}{f_{\text{khung}}} \tag{12.1}$$

Trong công thức, Δv là độ lớn của bước nhảy giữa hai khung hình liên tiếp và f là tần số khung hình. Mỗi bước nhảy ấy là một chỗ gãy trong đường bao, đúng cái đã gây tiếng tách ở mục 6.6 — chỉ khác là ở đây nó lặp lại sáu mươi lần mỗi giây, thành một tiếng rè liên tục mà người ta hay đổ cho bộ lọc.

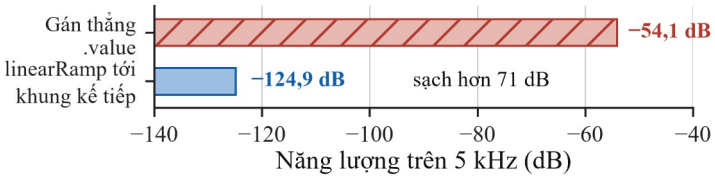
Bảng 12.2 Ba nhịp thời gian cùng chạy trong một ứng dụng

Nhịp	Tần suất	Khoảng cách	Ai điều khiển
Khung hình giao diện	60 lần/giây	16,7 ms	Trình duyệt, theo màn hình
Khối dựng âm thanh	344,53 lần/giây	2,90 ms	Web Audio, cố định 128 mẫu
Mẫu	44.100 lần/giây	22,7 µs	Phần cứng

Bảng cho thấy gốc của vấn đề bằng con số: giữa hai lần giao diện kịp viết vào tham số, Web Audio đã dựng xong gần sáu khối. Sáu khối ấy nhận đúng một giá trị không đổi, rồi khối thứ bảy nhận một giá trị khác hẳn.



(a) Cùng một cử chỉ, hai cách viết vào tham số



(b) Phần năng lượng trên 5 kHz trong tiếng ra

Hình 12.3 Nhiễu răng cưa của phép gán thẳng, và cách sửa bằng một dòng

12.2.2 Cách sửa: ramp tới khung hình kế tiếp

Cách sửa chỉ dài hơn một dòng. Thay vì gán thẳng, hãy hẹn cho tham số đi tới giá trị mới trong đúng khoảng thời gian tới khung hình sau.

Liệt kê 12.2 Cách viết đúng — và đây là mã đã chạy để đo ra con số 71 decibel

```
const KHUNG = 1 / 60;                    // 16,7 ms

slider.addEventListener("input", (e) => {
  const v = Number(e.target.value);
  const t = ctx.currentTime;
  const p = gainNode.gain;
  p.cancelScheduledValues(t);
  p.setValueAtTime(p.value, t);          // NEO — xem B12.2.4
  p.linearRampToValueAtTime(v, t + KHUNG);
});
```

Kết quả đo trên tiếng ra: phần năng lượng trên 5 kilohertz hạ từ −54,1 xuống −124,9 decibel, tức sạch hơn 71 decibel. Cử chỉ vẫn theo tay đúng như cũ — người dùng không nhận ra gì khác ngoài việc tiếng rè biến mất.

> **Vì sao ramp tới khung sau chứ không phải 5 mili giây**
>
> Nếu ramp ngắn hơn khoảng cách hai khung hình thì tham số tới đích rồi đứng chờ — lại thành bậc thang, chỉ là bậc thang có góc bo.
>
> Nếu ramp dài hơn thì mỗi lệnh mới lại cắt ngang lệnh cũ giữa chừng, và cử chỉ bắt đầu bị trễ so với tay.
>
> Ramp đúng bằng một khung hình cho một đường liên tục tuyệt đối mà vẫn bám sát tay. Đây là lý do hằng số 1/60 xuất hiện trong mã, chứ không phải một con số chọn tuỳ ý.

12.2.3 Ba cách viết vào một AudioParam, và khi nào dùng cách nào

Web Audio cho ba cách đưa một tham số tới giá trị mới, và chúng khác nhau ở hình dạng đường đi chứ không ở đích đến.

Bảng 12.3 Ba cách đưa tham số tới giá trị mới

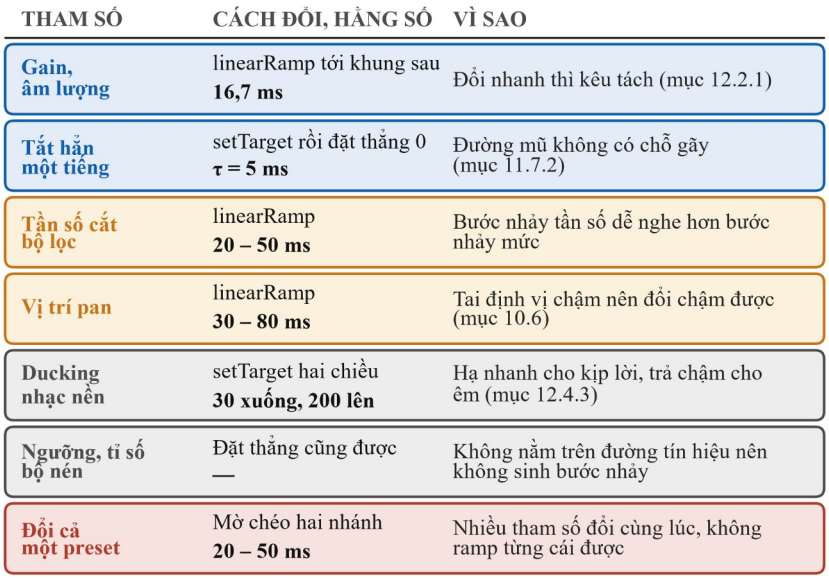
Cách gọi	Hình dạng	Tới đích khi nào	Dùng cho
setValueAtTime	Bước nhảy thẳng đứng	Ngay lập tức	Tham số ngoài đường tín hiệu; và để *neo*
linearRampToValueAtTime	Đường thẳng	Đúng thời điểm đã hẹn	Núm điều khiển, mờ chéo, pan
exponentialRampToValueAtTime	Đường mũ	Đúng thời điểm đã hẹn	Tần số và cao độ — tai nghe theo lô-ga
setTargetAtTime	Đường mũ tiệm cận	*Không bao giờ* tới hẳn	Ducking, tắt tiếng, đường bao trơn

Ba điểm dễ sai nhỏ của ba cách trên

exponentialRamp không đi qua được số 0 và cũng không xuất phát từ 0 — gọi với đích bằng 0 là lỗi. Muốn tắt hẳn bằng đường mũ thì ramp xuống một giá trị rất nhỏ rồi setValueAtTime(0) ngay sau.

setTargetAtTime nhận *hằng số thời gian* chứ không nhận thời điểm đích. Sau một hằng số, tham số đi được 63 phần trăm quãng đường; sau ba hằng số, 95 phần trăm. Muốn tắt hẳn thì vẫn phải đặt thẳng sau đó.

Cả ba đều tính từ giá trị đang có *tại thời điểm hẹn*, không phải giá trị lúc gọi hàm. Đây là nguồn của toàn bộ mục 12.2.4.

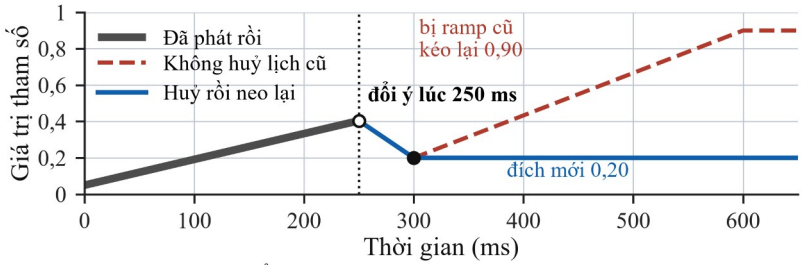


Hình 12.4 Mỗi loại tham số một cách đổi và một hằng số thời gian

12.2.4 Điểm dễ sai lớn nhất: bỏ sót việc huỷ lịch cũ

Người dùng kéo thanh trượt xuống rồi đổi ý và kéo ngược lên giữa chừng. Lệnh ramp thứ hai được gọi trong khi lệnh ramp thứ nhất còn chưa chạy xong. Chuyện gì xảy ra?

Câu trả lời làm nhiều người ngạc nhiên: cả hai lệnh cùng có hiệu lực. AudioParam giữ một *danh sách* sự kiện chứ không giữ một giá trị. Lệnh mới không thay thế lệnh cũ — nó xếp vào sau, và đường đi thật là kết quả của cả danh sách.



(a) Đổi ý giữa chừng một đường bao đang chạy

BƯỚC 1 — ĐỌC	`const v = p.value;  // giá trị hiện tại`
BƯỚC 2 — HUỶ	`p.cancelScheduledValues(t);`
BƯỚC 3 — NEO	`p.setValueAtTime(v, t);` `p.linearRampToValueAtTime(moi, t + fade);`

Thiếu bước 2 thì lệnh cũ còn hiệu lực và kéo tham số ngược lại.
Thiếu bước 3 thì tham số nhảy về giá trị cũ.

(b) Ba dòng đi liền nhau mỗi lần lập lịch mới

Hình 12.5 Quên huỷ lịch cũ, và cách huỷ rồi neo lại cho đúng

Trong phép đo của hình, một ramp dài đang đưa tham số lên 0,9 thì bị một ramp mới cắt ngang, hạ xuống 0,2. Bản không huỷ chạm 0,2 đúng như hẹn rồi bị ramp cũ kéo ngược trở lại 0,9. Bản huỷ đúng cách dừng ở 0,2 và giữ nguyên.

Liệt kê 12.3 Ba dòng phải đi liền nhau mỗi lần lập lịch mới

```
// SAI — chỉ huỷ mà không neo lại
p.cancelScheduledValues(t);
p.linearRampToValueAtTime(v, t + 0.05);

// ĐÚNG — huỷ, NEO giá trị hiện tại, rồi mới ramp
p.cancelScheduledValues(t);
p.setValueAtTime(p.value, t);
p.linearRampToValueAtTime(v, t + 0.05);
```

Dòng neo ở giữa là dòng người ta hay bỏ. Không có nó, ramp mới không biết xuất phát từ đâu: cancelScheduledValues xoá cả điểm mốc cuối cùng, nên tham số rơi về giá trị của sự kiện còn lại gần nhất trước đó — thường là một chỗ hoàn toàn không liên quan.

Một ghi chú trung thực về cancelAndHoldAtTime

Đặc tả có sẵn một hàm làm đúng việc huỷ-và-neo trong một lời gọi: cancelAndHoldAtTime. Trên giấy nó là câu trả lời gọn hơn.

Trên bản cài đặt dùng để dựng cuốn sách này, cancelAndHoldAtTime cho kết quả hoàn toàn giống cancelScheduledValues — nghĩa là nó *không* neo. Vì vậy bài này dạy mẫu ba dòng, là mẫu chạy đúng ở mọi nơi.

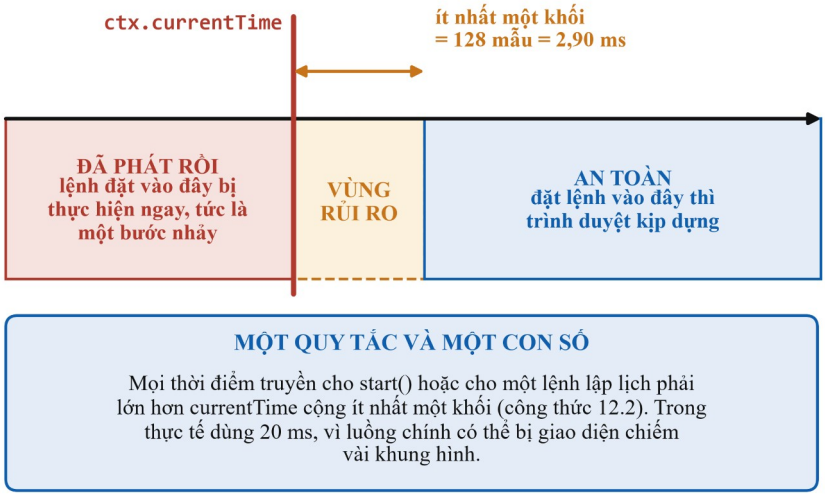
Người đọc nên tự kiểm hàm ấy trên trình duyệt đích của mình trước khi dùng. Cách kiểm nằm ngay trong bài tập 3 cuối chương.

12.2.5 Lập lịch an toàn: đừng bao giờ hẹn vào hiện tại

Một lỗi nữa rất hay gặp và rất khó thấy: hẹn sự kiện vào đúng currentTime. Trông thì hợp lý — hẹn vào bây giờ. Nhưng currentTime là thời điểm *đầu* của khối đang dựng, và khi mã của chúng ta chạy thì khối ấy có thể đã dựng xong một phần.

$$t_{\text{lệnh}} \geq t_{\text{hiệntại}} + t_{\text{antoàn}}, \qquad t_{\text{antoàn}} \geq \frac{N_q}{f_s} \tag{12.2}$$

Trong công thức, N là số mẫu của một khối dựng — bằng 128 — và f là tần số lấy mẫu. Với 44.100 hertz, khoảng an toàn tối thiểu là 2,90 mili giây. Thực tế nên lấy rộng hơn: 20 mili giây là con số quen dùng, đủ để chịu được một lần trễ luồng mà vẫn không ai nghe ra độ trễ.



Hình 12.6 Vì sao mọi lệnh lập lịch đều cộng thêm một khoảng an toàn

Bảng 12.4 Khoảng an toàn nên dùng trong bốn tình huống

Tình huống	Khoảng an toàn	Vì sao
Người dùng vừa bấm nút	20 ms	Không ai nhận ra 20 ms; và nó chịu được một lần trễ luồng
Nốt nhạc theo bộ lập lịch	100 – 200 ms	Cửa sổ nhìn trước của mục 11.7.4 đã lo phần này
Đổi tham số theo cử chỉ	20 ms	Đủ an toàn mà vẫn bám tay — xem mục 12.2.2
Tiếng phản hồi tức thì trong trò chơi	0 ms, dùng start()	Gọi start() không tham số là lệnh phát ngay ở khối kế tiếp

Bốn việc cần nhớ của mục 12.2

Không bao giờ gán thẳng vào .value của một tham số nằm trên đường tín hiệu. Luôn ramp — dù chỉ ramp một khung hình.

Mỗi lần lập lịch mới: huỷ, *neo*, rồi mới ramp. Ba dòng, luôn đi cùng nhau.

Mọi thời điểm hẹn đều là currentTime cộng một khoảng an toàn, không bao giờ là currentTime trần.

Gói cả ba việc trên vào một hàm dat() duy nhất và bắt mọi núm đi qua nó — xem mã ở mục 12.7.3. Đây là cách duy nhất để không bỏ sót ở lần thứ ba mươi.

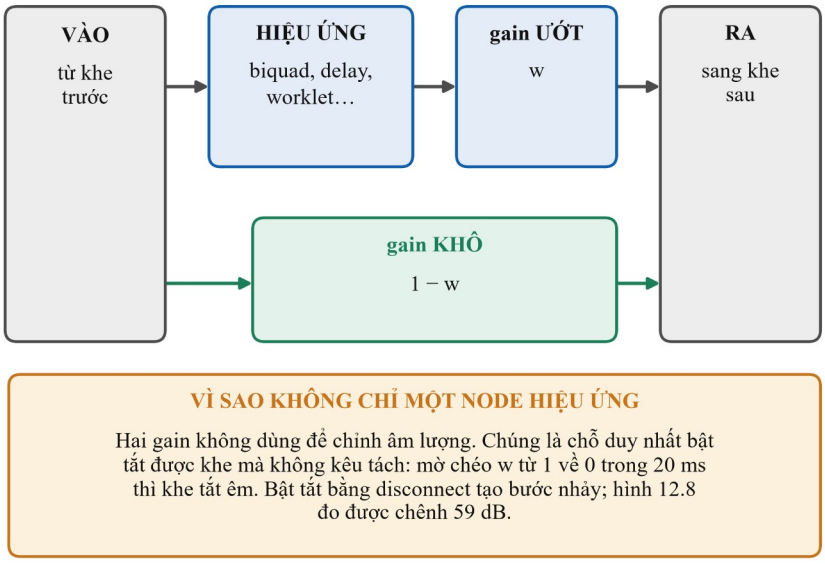
12.3 Rack hiệu ứng — chuỗi khối bật tắt được

12.3.1 Một rack là gì, và vì sao nó không chỉ là một chuỗi

Chương 11 đã nối các node thành chuỗi bằng connect. Một rack hiệu ứng là chuỗi ấy cộng thêm ba thứ mà một ứng dụng thật nào cũng cần: mỗi khối bật tắt được riêng, mỗi khối trộn được khô với ướt, và thứ tự các khối đổi được.

Bảng 12.5 Ba thứ một rack có mà một chuỗi connect thường không có

Thứ	Cần gì thêm	Mục bàn
Bật tắt từng khe	Hai nhánh song song và hai GainNode để mờ chéo giữa chúng	12.3.2
Trộn khô với ướt	Một GainNode trên mỗi nhánh, và một luật trộn đúng	12.3.4
Đổi thứ tự khe	Ngắt và nối lại toàn chuỗi — và biết trước hậu quả của thứ tự	12.3.5



Hình 12.7 Kiến trúc một rack: khe, nhánh bypass, và đường khô song song

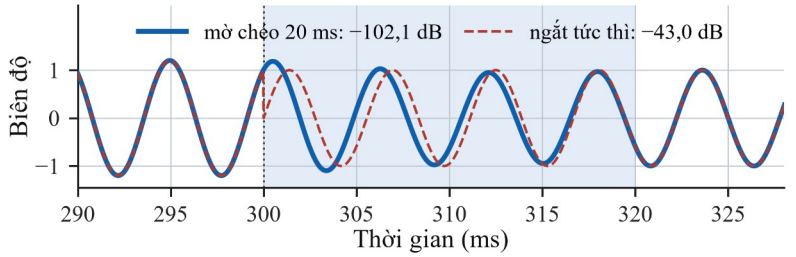
$$C_{\text{rack}} = \sum_{i=1}^{M} b_i c_i, \qquad b_i \in \{0,\ 1\} \tag{12.3}$$

Công thức nói một điều đơn giản nhưng hay bị bỏ qua: chi phí của cả rack là tổng chi phí của những khe *đang bật*, với b là cờ bật tắt của khe thứ i và c là chi phí của nó. Câu hỏi thú vị nằm ở chỗ: khi b bằng 0 thì cái gì làm cho c biến mất khỏi tổng?

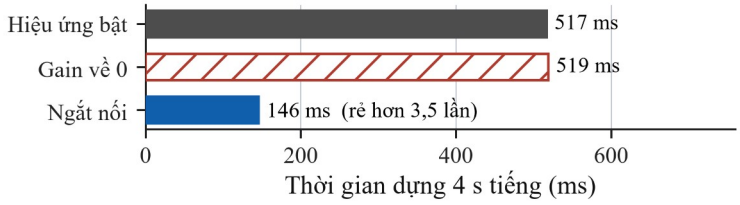
12.3.2 Bypass: hai cách làm, và chỉ một cách nghe được

Cách thứ nhất là ngắt nối: gọi disconnect trên khối cần tắt rồi nối thẳng đầu vào sang đầu ra. Cách thứ hai là mờ chéo: dựng sẵn hai nhánh song song, một nhánh có hiệu ứng và một nhánh không, rồi chuyển giữa hai gain của chúng.

Cách thứ nhất là cách ai cũng nghĩ tới, và nó tạo ra một bước nhảy ngay tại điểm ngắt. Đo trên tiếng ra: năng lượng cao tần quanh chỗ chuyển là −43,0 decibel với cách ngắt tức thì, so với −102,1 với mờ chéo 20 mili giây. Chênh nhau 59 decibel — và 59 decibel là khoảng cách giữa một tiếng bụp rõ mồn một và không nghe thấy gì.



(a) Dạng sóng quanh chỗ chuyển và năng lượng cao tần



(b) Chi phí CPU của mỗi cách

Hình 12.8 Hai cách bypass, và cái giá tính bằng CPU của mỗi cách

Liệt kê 12.4 Bypass đúng cách: mờ chéo trước, ngắt nối sau

```
batTat(khe, bat, ms = 20) {
  const t = this.ctx.currentTime + 0.02;
  const g = ms / 1000;
  for (const [p, dich] of [[khe.uotGain.gain, bat ? 1 : 0],
                           [khe.khoGain.gain, bat ? 0 : 1]]) {
    p.cancelScheduledValues(t);
    p.setValueAtTime(p.value, t);
    p.linearRampToValueAtTime(dich, t + g);
  }
  if (!bat) {
    setTimeout(() => khe.uot.disconnect(), ms + 60);
  } else if (!khe.dangNoi) {
    khe.uot.connect(khe.uotGain); khe.dangNoi = true;
  }
```

```
}
```

12.3.3 Và bây giờ là con số làm nhiều người bất ngờ

Câu hỏi: nếu chỉ cần đặt gain của nhánh hiệu ứng về 0 là đủ tắt tiếng, thì việc gì phải ngắt nối cho mất công?

Phép đo trả lời dứt khoát. Cùng một đồ thị hai mươi bốn giọng, dựng bốn giây tiếng: bật hiệu ứng mất 517 mili giây; đặt gain về 0 mất 519 mili giây; ngắt nối mất 146 mili giây.

> **Đặt gain về 0 không tiết kiệm gì cả**
>
> 519 so với 517 mili giây — chênh lệch nằm trong sai số phép đo. Web Audio vẫn chạy đầy đủ mọi node trong nhánh, rồi mới nhân kết quả với 0 ở cuối.
>
> Chỉ có ngắt nối mới làm cho nhánh biến mất khỏi đồ thị dựng, và lúc ấy mới rẻ — 146 so với 517, tức 3,5 lần.
>
> Kết luận thực hành: mờ chéo về 0 trước cho khỏi kêu, *rồi* mới ngắt nối sau khi mờ xong. Hai việc, hai lý do khác nhau, và bỏ việc nào cũng sai một nửa.

12.3.4 Trộn khô với ướt, và một luật hay bị chọn sai

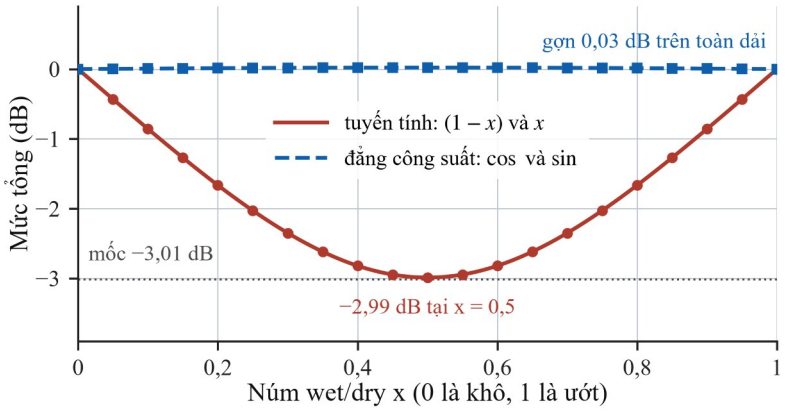
Mỗi khe trong rack có một núm wet/dry quyết định bao nhiêu phần tín hiệu đi qua hiệu ứng. Cách đơn giản nhất là chia tuyến tính: nhánh khô nhân với 1 trừ x, nhánh ướt nhân với x. Cách này sai — và sai theo đúng kiểu đã gặp ở mục 10.4 khi bàn về luật pan.

$$L = 10\log_{10}\left(g_k^2 + g_u^2 + 2\rho\, g_k g_u\right) \tag{12.4}$$

Mức tổng hợp phụ thuộc hệ số tương quan ρ giữa hai nhánh. Với hai tín hiệu hoàn toàn giống nhau, ρ bằng 1 và luật tuyến tính giữ được mức. Nhưng nhánh ướt là tiếng đã qua hiệu ứng nên nó gần như không tương quan với nhánh khô: ρ về gần 0, và mức tụt đúng ở giữa.

$$g_k = \cos\frac{\pi x}{2},\quad g_u = \sin\frac{\pi x}{2};\qquad x \in [0,\ 1] \tag{12.5}$$

Luật đẳng công suất dùng cosin và sin thay cho phép chia tuyến tính. Đo trên đồ thị chạy thật với một tiếng vang thật: luật tuyến tính hõm 2,99 decibel tại x bằng 0,5, còn luật đẳng công suất gọn tổng cộng 0,03 decibel trên toàn dải.



(a) Mức tổng theo hai luật trộn

Nhánh ướt đã qua hiệu ứng nên gần như không tương quan với nhánh khô (ρ gần 0): mức tổng cộng theo công suất, đúng công thức (12.4).

Hình 12.9 Hai luật trộn khô-ướt, và mức đo được trên toàn dải núm

Ba con số của mục này đều là 3 decibel — và chúng là một

Mục 10.2: cộng hai nguồn không tương quan thì mức lên 3,01 decibel chứ không phải 6,02.

Mục 10.4: luật pan tuyến tính hụt 3,01 decibel ở giữa.

Mục này: luật wet/dry tuyến tính hõm 2,99 decibel ở giữa.

Ba mục khác nhau, cùng một phép toán: cộng hai thứ không tương quan thì cộng theo công suất, không cộng theo biên độ.

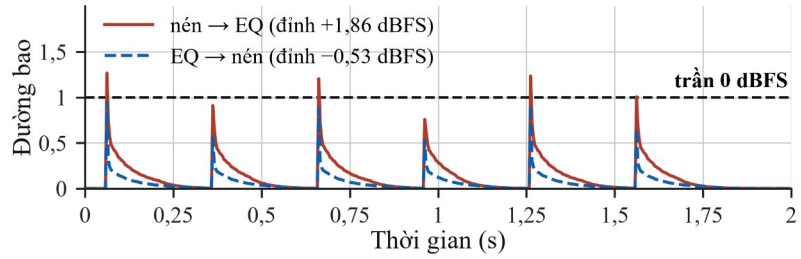
12.3.5 Thứ tự các khe — một quyết định, không phải một thói quen

Đặt bộ EQ trước hay sau bộ nén? Câu hỏi này trong phòng thu thường được trả lời bằng thói quen. Nhưng nó có một câu trả lời đo được, và câu trả lời ấy khác nhau tuỳ mục đích.

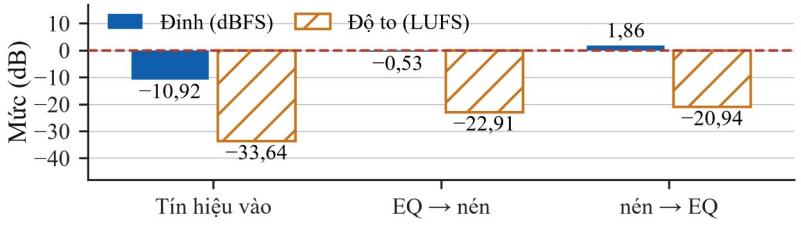
Phép đo dùng cùng một tiếng trống, một bộ EQ nâng 9 decibel ở 900 hertz, và một bộ nén ngưỡng −24 với tỉ số 6. Đổi thứ tự hai khối không đổi một chữ nào trong mã ngoài thứ tự hai lời gọi connect.

Bảng 12.6 Cùng hai khối, đổi thứ tự, và hai con số đổi hẳn

Cấu hình	Đỉnh (dBFS)	Độ to (LUFS)	Nhận xét
Tín hiệu vào	−10,92	−33,64	Chưa qua khối nào
EQ → nén	−0,53	−22,91	Bộ nén nhìn thấy phần đã nâng và kiểm soát được nó
nén → EQ	+1,86	−20,94	Phần nâng đi thẳng ra ngoài — *vượt trần* 0 dBFS



(a) Đường bao tín hiệu theo thứ tự hai khối



(b) Đỉnh và độ to theo cấu hình

Hình 12.10 Đổi thứ tự hai khối trong rack, và hai con số đổi theo

Đặt EQ trước nén thì bộ nén làm được việc của nó: đỉnh dừng ở −0,53 dBFS, dưới trần. Đặt EQ sau nén thì phần nâng đi thẳng ra ngoài, không ai kiểm soát, và đỉnh vọt lên +1,86 dBFS — tức đã cắt méo trước khi ra tới loa.

Thứ tự mặc định nên dùng khi chưa có lý do khác

Cổng và lọc bỏ tiếng ồn *đầu tiên* — không nên để bộ nén khuếch đại tiếng ồn nền lên rồi mới đi lọc.

EQ cắt trước bộ nén, EQ nâng cũng trước bộ nén — để bộ nén nhìn thấy đúng thứ sẽ ra ngoài.

Bộ nén rồi tới hiệu ứng thời gian — vang và trễ nên ăn vào tín hiệu đã ổn định mức, không thì đuôi vang cũng bị nén theo và nghe như đang thở.

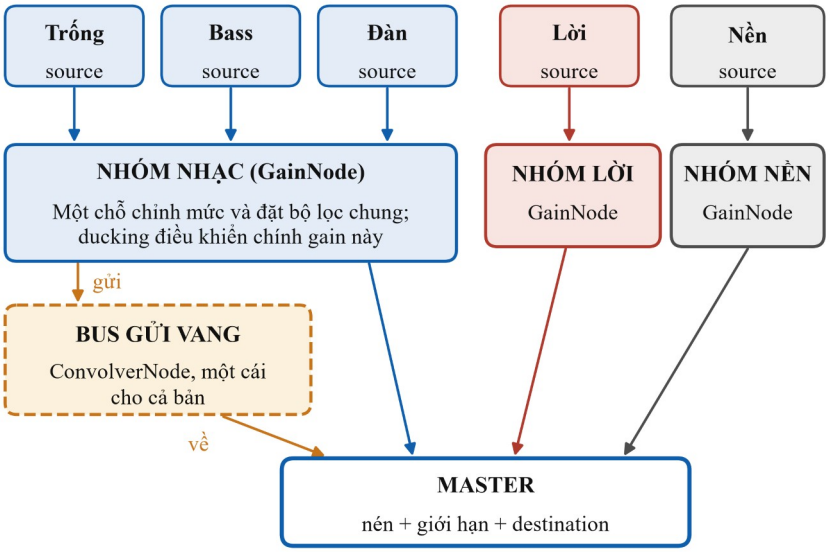
Bộ giới hạn *cuối cùng*, sau tất cả. Đây là mục 8.6, và nó là khối duy nhất có quyền nói câu cuối về đỉnh.

12.4 Trộn động — khi bản trộn tự đổi theo sự kiện

12.4.1 Bus và nhóm: chỗ ổn định để gắn mọi thứ vào

Trong một ứng dụng thật, nguồn phát sinh ra và chết đi liên tục: mỗi bước chân là một BufferSourceNode mới, mỗi nốt nhạc là một giọng mới. Không thể gắn núm âm lượng vào một thứ chết sau nửa giây.

Giải pháp là chèn một tầng ở giữa: mỗi loại tiếng đi qua một GainNode dựng sẵn lúc nạp ứng dụng và sống suốt phiên. Tầng ấy gọi là nhóm, và nó là chỗ duy nhất đáng gắn núm, bộ lọc chung, và ducking vào.



Hình 12.11 Ba tầng của một bản trộn: nguồn, nhóm, và bus gửi

Bảng 12.7 Bộ nhóm tối thiểu cho một ứng dụng trò chơi hoặc phát thanh

Nhóm	Chứa gì	Núm riêng	Việc đặc biệt
Nhạc	Nhạc nền, nhạc chuyển cảnh	Có	Là nhóm bị ducking hạ xuống
Lời	Thoại, thuyết minh	Có	Là nhóm kích hoạt ducking; thường không gửi vang
Hiệu ứng	Tiếng va chạm, bước chân, giao diện	Có	Nhóm đông nhất — chỗ đặt trần đa âm
Môi trường	Tiếng nền liên tục: gió, mưa, đám đông	Có	Ít giọng nhưng dài; gửi vang nhiều nhất
Master	Tất cả bốn nhóm trên	Có	Chỗ đặt bộ giới hạn cuối cùng — mục 8.6

Phân biệt nhóm với bus gửi

Nhóm nằm *trên* đường tín hiệu: mọi thứ đi qua nó rồi mới ra tiếp. Tắt nhóm là tắt cả loại tiếng ấy.

Bus gửi nằm *song song*: mỗi nguồn gửi một phần tín hiệu sang, kết quả cộng thêm vào bản trộn ở master. Tắt bus gửi thì tiếng vẫn còn, chỉ mất phần vang.

Nhầm hai thứ này là nguồn của lỗi điển hình: đặt convolver *trên* đường tín hiệu thay vì song song, rồi không hiểu vì sao không tắt được tiếng khô đi đâu cả.

12.4.2 Ducking — nhạc nền tự nhường chỗ cho lời

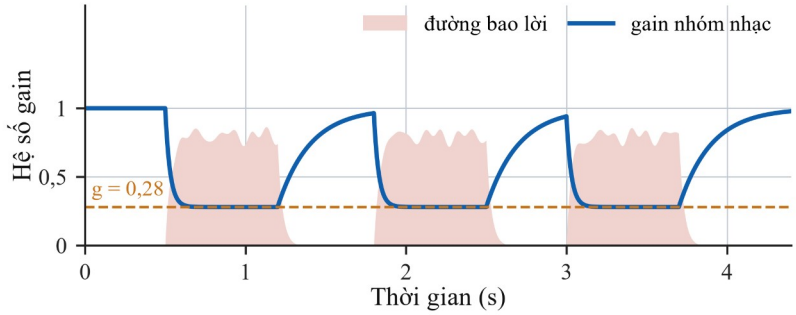
Ducking là việc hạ mức một nhóm xuống trong lúc một nhóm khác đang phát. Trong trò chơi và trong phát thanh, nó là thứ khiến người nghe không phải tự chỉnh âm lượng mỗi khi có lời thoại.

$$g(t) = 1 - (1 - g_{\min})\, d(t), \qquad d(t) \in [0,\ 1] \tag{12.6}$$

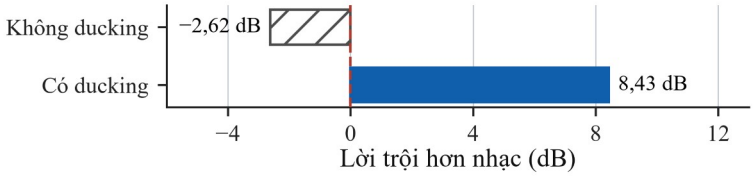
Trong công thức, d là tín hiệu điều khiển đi từ 0 tới 1 theo mức độ có tiếng ở nhóm ưu tiên, còn g là mức thấp nhất mà nhóm bị hạ xuống tới. Với g bằng 0,28 như trong phép đo, nhóm nhạc bị hạ 11,1 decibel khi có lời.

$$\Delta_{\text{lời}} = L_{\text{lời}} - L_{\text{nhạc}}\ ; \qquad \Delta_{\text{lời}}^{\text{duck}} = \Delta_{\text{lời}} - 20\log_{10} g_{\min} \tag{12.7}$$

Công thức thứ hai nói phần thu được: chênh lệch giữa lời và nhạc tăng thêm đúng bằng lượng đã hạ. Đo lại trên tiếng ra thật — không phải trên đường điều khiển — cho kết quả: không ducking thì lời *thấp* hơn nhạc 2,62 decibel; có ducking thì lời *trội* hơn nhạc 8,43 decibel. Đổi 11,06 decibel, khớp với lượng hạ đã đặt.



(a) Gain nhóm nhạc theo ba câu lời: hạ τ = 30 ms, trả τ = 200 ms



(b) Mức lời so với nhạc trên tiếng ra, đổi 11,06 dB

Hình 12.12 Đường điều khiển ducking, và lời nổi lên bao nhiêu trên tiếng ra

Liệt kê 12.5 Ducking hai chiều: hạ nhanh, trả chậm

```
const TAU_XUONG = 0.030;    // 30 ms – kịp trước âm tiết đầu
const TAU_LEN   = 0.200;    // 200 ms – trả chậm cho êm
const G_MIN     = 0.28;

duck(bat) {
  const p = this.nhacGain.gain;
  const t = this.ctx.currentTime + 0.02;
  p.cancelScheduledValues(t);
  p.setValueAtTime(p.value, t);          // NEO – bắt buộc
  p.setTargetAtTime(bat ? G_MIN : 1,
                    t, bat ? TAU_XUONG : TAU_LEN);
}
```

Hai hằng số không bao giờ bằng nhau

Hạ phải *nhanh*, vì tai nghe thấy ngay khi lời bị nhạc lấn. 20 tới 40 mili giây là khoảng quen dùng.

Trả phải *chậm*, vì trả nhanh làm nhạc phồng lên giữa hai câu, nghe như đang bơm. 150 tới 400 mili giây tuỳ nhịp lời.

Đặt hai hằng số bằng nhau là lỗi hay gặp nhất, và triệu chứng của nó rất dễ nhận: nhạc nền thở theo từng câu.

Và dòng setValueAtTime neo lại trong mã trên *không* được bỏ. Không có nó, setTargetAtTime tính từ một giá trị không xác định — trong phép thử đầu tiên khi dựng bài này, nó cho ra những con số như 2.136.643,5.

12.4.3 Lấy tín hiệu điều khiển từ đâu

Có ba cách lấy tín hiệu d trong công thức (12.6), và chúng khác nhau về độ chính xác lẫn độ phức tạp.

Bảng 12.8 Ba cách sinh tín hiệu điều khiển cho ducking

Cách	Cách làm	Ưu	Nhược
Theo sự kiện	Gọi duck(true) khi bắt đầu phát lời, duck(false) khi lời hết	Đơn giản, chính xác tuyệt đối, không tốn gì	Chỉ dùng được khi biết trước lời bắt đầu lúc nào
Theo đường bao	Chạy AnalyserNode trên nhóm lời, đọc mức mỗi khung hình rồi lái gain	Theo được tiếng trực tiếp, ví dụ micrô	Trễ một khung hình; phải tự làm ngưỡng và trễ nhả
Bằng sidechain	Cho nhóm lời điều khiển một DynamicsCompressor đặt trên nhóm nhạc	Đúng kiểu phòng thu, ổn định nhất	Web Audio *không* có đầu vào sidechain — phải tự dựng bằng worklet

Với ứng dụng trò chơi và ứng dụng phát thanh có kịch bản, cách thứ nhất là cách nên dùng — nó không tốn gì và không sai bao giờ. Hai cách sau chỉ cần tới khi nguồn ưu tiên là tiếng trực tiếp mà ứng dụng không biết trước.

Sự kiện	Đổi tham số nào	Đổi ra sao	Điểm dễ sai
Lời bắt đầu	gain nhóm nhạc	`setTarget` `τ = 30 ms`	Bỏ sót việc trả lại khi lời im
Lời kết thúc	gain nhóm nhạc	`setTarget` `τ = 200 ms`	Trả nhanh quá: nhạc nghe như bơm
Vào phòng khác	mức gửi tới bus vang	`linearRamp` `300 ms`	Đổi buffer vang giữa chừng: kêu tách
Nhân vật lại gần	gain nguồn và pan	`linearRamp` `80 ms`	Ramp pan nhanh: tiếng nhảy trong đầu
Sức khoẻ xuống thấp	bộ lọc thấp qua trên nhóm	`ramp fc` `500 ms`	Đổi type bộ lọc lúc đang chạy
Tạm dừng trò chơi	gain master	`setTarget` `τ = 15 ms`	suspend() tức thì cũng kêu tách
Chuyển cảnh	mờ chéo hai nhánh	`400 – 800 ms`	Bỏ sót việc ngắt nhánh cũ sau khi mờ xong

Mọi điểm dễ sai trong bảng đều liên quan đến thời gian, không phải đến giá trị.

Hình 12.13 Sự kiện nào đổi tham số nào, đổi ra sao, và điểm dễ sai đi kèm

12.4.4 Một quy tắc để không loạn

Tới đây một ứng dụng đã có khá nhiều thứ cùng muốn viết vào cùng một chỗ: người dùng kéo núm, ducking hạ nhóm nhạc, khoảng cách nhân vật đổi mức, và trạng thái trò chơi đổi bộ lọc. Bốn thứ ấy chạy độc lập với nhau và không thứ nào biết ba thứ kia đang làm gì.

> **Mọi tham số động đều phải có một chủ duy nhất**
>
> Khi ducking hạ nhóm nhạc xuống 0,28 và người dùng cùng lúc kéo núm âm lượng nhạc, hai bên cùng ghi vào một AudioParam — và bên gọi sau thắng, bất kể ý định.
>
> Cách sửa: tách làm hai GainNode nối tiếp. Một cái cho người dùng, một cái cho ducking. Mỗi tham số một chủ, và mức cuối là tích của hai — đúng thứ chúng ta muốn.
>
> Quy tắc này áp cho mọi tham số động, không riêng ducking: âm lượng theo khoảng cách, âm lượng theo trạng thái, và âm lượng do người dùng đặt là ba thứ khác nhau và nên nằm trên ba node khác nhau.

Chi phí của cách này rất nhỏ: mỗi GainNode thêm vào tốn khoảng 25 micro giây lúc dựng — xem mục 12.6.4 — và gần như không tốn gì lúc chạy, vì nhân với một hằng số là phép rẻ nhất trong cả đồ thị. Đổi lại, mã đọc được và không bao giờ có hai chỗ cùng ghi vào một tham số.

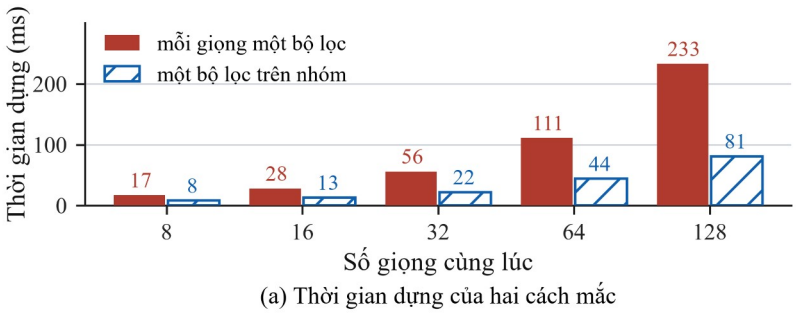
12.5 Tối ưu tài nguyên — làm ít đi chứ không làm nhanh hơn

12.5.1 Dùng chung một bộ lọc thay vì mỗi giọng một bộ

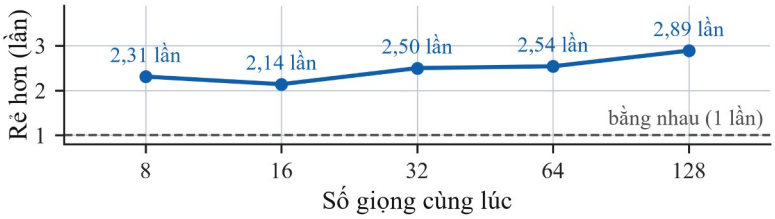
Khi ba mươi giọng cùng cần một bộ lọc thấp qua với cùng tần số cắt, có hai cách mắc. Cách thứ nhất: mỗi giọng một BiquadFilterNode riêng. Cách thứ hai: một bộ lọc duy nhất đặt trên GainNode của nhóm, sau chỗ tất cả đã cộng lại.

$$C_{\text{riêng}} = N(c_{\text{giọng}} + c_{\text{hiệuứng}}) \; ; \qquad C_{\text{chung}} = N\,c_{\text{giọng}} + c_{\text{hiệuứng}} \tag{12.8}$$

Công thức viết ra sự khác biệt: cách riêng nhân cả chi phí hiệu ứng lên N lần, cách chung thì không. Vì bộ lọc là tuyến tính và bất biến theo thời gian nên lọc tổng bằng đúng tổng các bản đã lọc — hai cách cho cùng một kết quả nghe được, chỉ khác giá.



(a) Thời gian dựng của hai cách mắc



(b) Tỉ số giữa hai cách, tăng theo số giọng

Hình 12.14 Cùng một tiếng, hai cách mắc bộ lọc, và tỉ số giữa hai cách

Đo trên đồ thị chạy thật: với 8 giọng, cách chung rẻ hơn 2,31 lần; với 128 giọng, 2,89 lần. Điều đáng chú ý là tỉ số ấy *không* cố định mà nhích lên theo số giọng — càng đông thì càng đáng gộp.

Bảng 12.9 Khối nào gộp chung được, khối nào không

Khối	Tuyến tính?	Gộp được?	Lý do
GainNode	Có	Được	Nhân với hằng số — cộng rồi nhân bằng nhân rồi cộng
BiquadFilterNode	Có	Được	Lọc tổng bằng tổng các bản lọc
DelayNode	Có	Được	Trễ tổng bằng tổng các bản trễ
ConvolverNode	Có	Được — và đây là chỗ lãi nhất	Chập tổng bằng tổng các bản chập — công thức (12.9)
DynamicsCompressorNode	Không	*Không*	Nén tổng khác tổng các bản nén — mục 8.4
WaveShaperNode	Không	*Không*	Méo tổng khác tổng các bản méo; sinh cả tần số tổng và hiệu
PannerNode	Không, vì mỗi giọng một vị trí	*Không*	Gộp là mất thông tin vị trí riêng — mục 10.7

Khi nào không được gộp

Khi mỗi giọng cần một thiết lập khác nhau — mỗi nhân vật một tần số cắt riêng theo khoảng cách chẳng hạn. Lúc ấy gộp là đổi kết quả nghe được, không phải tối ưu.

Khi hiệu ứng không tuyến tính. Bộ nén, bộ méo, bộ cắt đỉnh: nén tổng *khác* tổng các bản đã nén. Mục 8.4 đã cho thấy điều này.

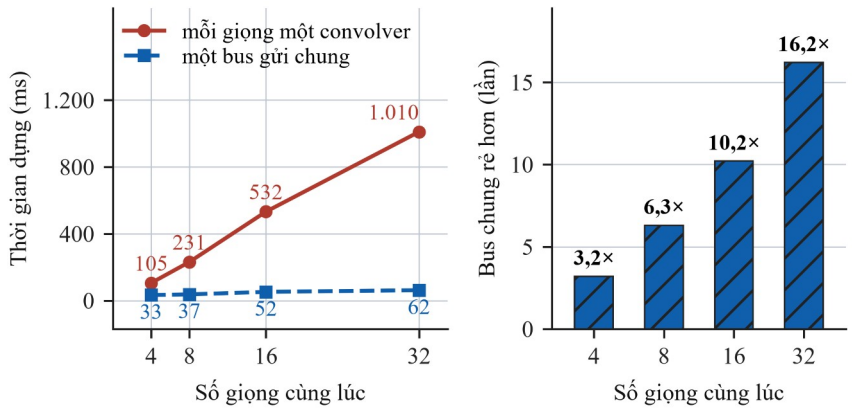
Quy tắc gọn: khối tuyến tính thì gộp được, khối phi tuyến thì không. Bộ lọc, gain, trễ, vang — gộp được. Nén, méo, giới hạn — không.

12.5.2 Bus gửi vang — chỗ tiết kiệm lớn nhất trong cả chương

Phép chập là phép tính đắt nhất trong một đồ thị âm thanh thông thường. Cho mỗi giọng một ConvolverNode riêng là cách chắc chắn nhất để làm một ứng dụng ngừng hoạt động.

$$y[n] = \sum_{i=1}^{N} x_i[n] + \left(h * \sum_{i=1}^{N} s_i x_i \right)[n] \qquad (12.9)$$

Công thức nói rõ vì sao gộp được: phép chập là tuyến tính, nên chập tổng đã trộn bằng đúng tổng các bản đã chập. Hệ số s là mức gửi của giọng thứ i — mỗi giọng vẫn có phần vang riêng nhiều ít khác nhau, nhưng phép chập chỉ chạy *một* lần.



(a) Thời gian dựng (b) Tỉ số chung/riêng

Hình 12.15 Cái giá của việc mỗi giọng mang một phòng vang riêng

Bảng 12.10 Vang riêng và vang chung — thời gian dựng đo được

Số giọng	Mỗi giọng một convolver	Một bus gửi chung	Chung rẻ hơn
4	105 ms	33 ms	3,2 lần
8	231 ms	37 ms	6,3 lần
16	532 ms	52 ms	10,2 lần
32	1.010 ms	62 ms	16,2 lần

Cột thứ hai tăng gần như tuyến tính theo số giọng. Cột thứ ba gần như đứng yên: từ 33 lên 62 mili giây khi số giọng tăng tám lần. Đó chính là hình dạng mà công thức (12.9) dự đoán, và nó là lý do mọi phần mềm trộn nhạc trên đời đều có khái niệm bus gửi.

Liệt kê 12.6 Bus gửi vang — dựng một lần, mọi giọng dùng chung

```
// LÚC NẠP ỨNG DỤNG — đúng một lần
this.vang = this.ctx.createConvolver();
this.vang.buffer = await this.napIR("phong_vua.wav");
this.vangTra = this.ctx.createGain();
this.vangTra.gain.value = 0.8;
this.vang.connect(this.vangTra).connect(this.master);

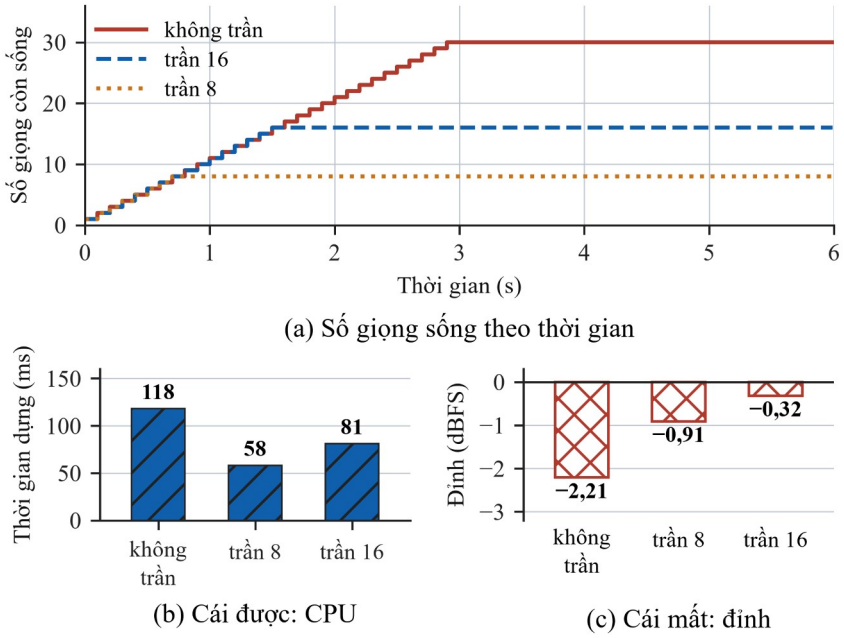
// MỖI GIỌNG — chỉ thêm một GainNode nhỏ
if (gui > 0) {
  const sg = this.ctx.createGain();
  sg.gain.value = gui;              // hệ số s trong (12.9)
  g.connect(sg).connect(this.vang);
}
```

12.5.3 Trần đa âm — cắt bớt giọng, và cái giá của việc cắt

Khi một mẫu trống chạy nhanh, đuôi của tiếng trước còn chưa tắt thì tiếng sau đã vào. Số giọng cùng sống cứ thế leo lên cho tới khi nó bằng số tiếng phát trong quãng thời gian bằng độ dài đuôi.

$$N_{\text{sóng}} = \min\left(N_{\text{trần}}, \left\lceil \frac{T_{\text{đuôi}}}{T_{\text{bước}}} \right\rceil\right) \tag{12.10}$$

Số giọng sống là giá trị nhỏ hơn giữa trần đã đặt và tỉ số giữa độ dài đuôi với khoảng cách hai lần phát. Vế thứ hai là con số tự nhiên của bản nhạc; vế thứ nhất là con số chúng ta đặt ra.



(a) Số giọng sống theo thời gian

(b) Cái được: CPU

(c) Cái mất: đỉnh

Hình 12.16 Trần đa âm: cái được tính bằng CPU và cái mất tính bằng decibel

Đo trên một mẫu 150 nhịp mỗi phút với sáu lớp: không đặt trần thì dựng mất 118 mili giây; đặt trần 8 còn 58; đặt trần 16 còn 81. Nhưng nhìn cột đỉnh thì thấy cái giá: cắt bớt giọng cũng cắt bớt phần cộng dồn, nên đỉnh đi từ −2,21 lên −0,91 dBFS. Tiếng nghe *thưa* hơn, không phải chỉ nhẹ máy hơn.

Liệt kê 12.7 Trần đa âm có mờ ra, không cắt thẳng

```
phatCoTran(ten, opts) {
  if (this.song.size >= TRAN) {
    const cu = this.song.keys().next().value;   // giọng già nhất
    this.dungMuot(cu, 0.015);                   // mờ 15 ms rồi stop
  }
  return this.phat(ten, opts);
}

dungMuot(id, ms) {
  const { s, g } = this.song.get(id);
  const t = this.ctx.currentTime + 0.005;
  g.gain.cancelScheduledValues(t);
  g.gain.setValueAtTime(g.gain.value, t);
  g.gain.linearRampToValueAtTime(0, t + ms);
  s.stop(t + ms + 0.005);      // onended sẽ tự dọn — xem B12.6
}
```

Chọn giọng nào để cắt

Cắt giọng *già nhất* là mặc định hợp lý và là thứ mã trên làm. Nó gần đúng với chỗ tai ít để ý nhất, vì giọng già thường đã ở phần đuôi.

Cắt giọng *nhỏ nhất* thì đúng hơn về tâm lý nghe nhưng tốn công theo dõi mức từng giọng — chỉ đáng làm khi trần thấp.

Không bao giờ cắt giọng mới nhất. Người dùng vừa bấm mà không nghe thấy gì là lỗi nghiêm trọng hơn nhiều so với việc mất một đuôi vang.

Và luôn mờ ra 10 tới 20 mili giây thay vì gọi stop thẳng — nếu không, trần đa âm tự nó trở thành máy sinh tiếng tách.

12.5.4 Thứ tự làm việc khi tối ưu

Bốn mục vừa rồi cho bốn cách tiết kiệm với bốn tỉ số rất khác nhau. Xếp chúng theo tỉ số được trên công bỏ ra thì thứ tự không giống thứ tự người ta hay nghĩ tới.

VIỆC	ĐO ĐƯỢC BAO NHIÊU	CÁI GIÁ PHẢI TRẢ
1. Bus gửi chung cho vang	16,2 lần ở 32 giọng	Mọi giọng dùng chung một phòng vang
2. Ngắt nối nhánh không dùng	3,5 lần so với đặt gain bằng 0	Phải mờ chéo trước khi ngắt (mục 12.3.2)
3. Lọc chung theo nhóm	2,89 lần ở 128 giọng	Mất khả năng chỉnh bộ lọc riêng từng giọng
4. Trần đa âm	2,0 lần khi trần bằng 8	Tiếng thưa đi; nghe rồi hãy chọn
5. Dựng sẵn đồ thị cố định	Mỗi node ≈ 25 µs (công thức 12.11)	Tốn bộ nhớ thường trực, ít khi đáng lo
6. Đổi thuật toán bộ lọc	Chưa đo, và thường không đáng	Tốn nhiều công nhất, được ít nhất

Xếp theo tỉ số được trên công bỏ ra, không theo thứ tự thường nghĩ tới trước.

Dòng 1 và 2 cho tỉ số cao nhất và đo được ngay trong chương này.

Hình 12.17 Sáu việc tối ưu xếp theo tỉ số được trên công bỏ ra

Một câu cần nói rõ về những con số này

Mọi tỉ số trong mục 12.5 đo bằng thời gian *dựng ngoại tuyến*, tức thời gian máy cần để tính xong một đoạn tiếng — không phải thời gian thực.

Ưu điểm của cách đo ấy: nó lặp lại được, không phụ thuộc tải hệ thống, và so sánh được giữa hai lần chạy. Đó là lý do bài này dùng nó.

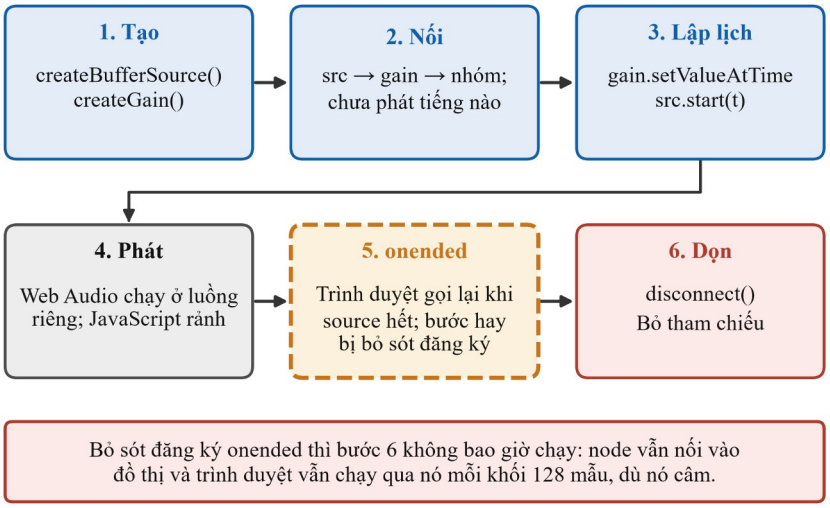
Giới hạn của nó: nó *không* cho biết ứng dụng chạy nổi bao nhiêu giọng trên máy người dùng. Muốn biết điều đó thì phải đo renderCapacity trên chính máy ấy, đúng như đã bàn ở mục 11.5.3.

Tỉ số thì chuyển được giữa hai cách đo; con số tuyệt đối thì không.

12.6 Vòng đời và bộ nhớ — node đã chết đi đâu

12.6.1 Sáu bước của một giọng

Mục 11.3.3 đã nói một node nguồn chỉ dùng được một lần. Trong một ứng dụng, điều đó có nghĩa là mỗi tiếng phát ra là một vòng đời đầy đủ, và vòng đời ấy có sáu bước.




```
src.onended = () => {
  src.disconnect(); gain.disconnect(); song.delete(id);
};
```

Hình 12.18 Sáu bước của một giọng, và bước duy nhất người ta hay bỏ sót

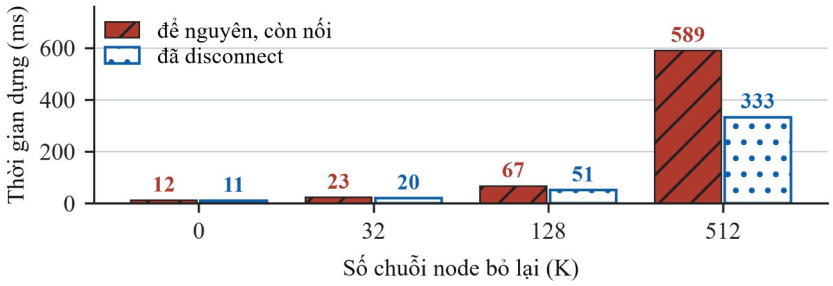
Bảng 12.11 Sáu bước, ai gọi, và chuyện gì sai nếu bỏ

Bước	Ai gọi	Bỏ thì sao
1. Tạo node	Mã của chúng ta	Không có tiếng
2. Nối vào đồ thị	Mã của chúng ta	Node chạy nhưng tiếng không tới đâu cả
3. Lập lịch tham số	Mã của chúng ta	Tiếng vào đột ngột — kêu tách, xem mục 12.2
4. Phát	Web Audio, ở luồng riêng	Không có gì sai — đây là bước chúng ta không đụng vào
5. onended	*Trình duyệt* gọi ngược lại mã của chúng ta	Bỏ sót đăng ký thì bước 6 không bao giờ chạy
6. Ngắt nối và bỏ tham chiếu	Mã của chúng ta, trong bước 5	Rò node: chậm dần và tốn bộ nhớ — mục 12.6.2

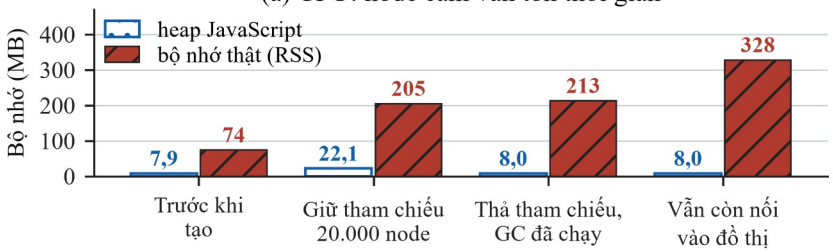
Năm bước đầu do mã của chúng ta gọi. Bước thứ sáu — dọn dẹp — cũng do mã của chúng ta gọi, nhưng gọi trong một hàm mà *trình duyệt* kích hoạt chứ không phải chúng ta. Đó là lý do nó hay bị bỏ sót: không có dòng nào trong mã tuyến tính nhắc tới nó.

12.6.2 Node bị bỏ sót không chỉ tốn bộ nhớ — nó tốn CPU

Đây là chỗ trực giác hay sai. Người ta nghĩ một node đã phát xong thì nó im, mà im thì không tốn gì. Web Audio không làm việc như vậy: chừng nào node còn nối vào đồ thị, trình duyệt còn phải chạy qua nó mỗi khối 128 mẫu.



(a) CPU: node câm vẫn tốn thời gian



(b) Bộ nhớ: bộ đo JavaScript không thấy

Hình 12.19 Cái giá của node bị bỏ sót, đo bằng CPU và bằng bộ nhớ

Bảng 12.12 Bỏ lại K chuỗi node đã phát xong — thời gian dựng đo được

Số chuỗi bỏ lại	Còn nối vào đồ thị	Đã disconnect	Chậm hơn
0	12 ms	11 ms	1,05 lần
32	23 ms	20 ms	1,13 lần
128	67 ms	51 ms	1,31 lần
512	589 ms	333 ms	1,77 lần

Tỉ số tăng theo số node bỏ lại, đúng như dự đoán. Nhưng cần đọc con số này cho đúng: 1,77 lần chứ không phải vô hạn. Một node câm vẫn rẻ hơn một node đang kêu, nên tỉ số dừng lại ở đó chứ không lớn thêm. Nói cách khác, node rò làm ứng dụng chậm dần chứ không làm nó chết ngay — và đó chính là lý do lỗi này sống sót lâu trong các dự án thật.

12.6.3 Vì sao công cụ đo bộ nhớ của trình duyệt không thấy

Phép đo bộ nhớ đi kèm cho một kết quả đáng để dừng lại nghĩ. Tạo 20.000 GainNode và giữ tham chiếu: heap JavaScript lên từ 7,9 tới 22,1 MB. Thả tham chiếu rồi cho bộ thu gom rác chạy: heap tụt về 8,0 MB — sạch sẽ.

Nhưng bộ nhớ thật của tiến trình không quay lại, và khi 20.000 node vẫn còn nối vào đồ thị thì nó leo thêm 115 MB nữa *trong khi* heap JavaScript đứng yên ở 8,0. Phần thân thật của một AudioNode nằm ngoài heap JavaScript.

Đọc hai cột này cho đúng

Cột heap thấp *không* chứng minh là không rò. Nó chỉ chứng minh là phần vỏ JavaScript đã được thu hồi.

Cột bộ nhớ thật cao cũng *không* chứng minh là có rò — bộ cấp phát giữ lại vùng nhớ đã xin của hệ điều hành là chuyện bình thường.

Thứ thật sự chứng minh có rò là *đếm*: giữ một Map các giọng đang sống và kiểm tra rằng nó trở về rỗng khi mọi tiếng đã tắt. Đây là phép kiểm rẻ nhất và chắc nhất, và nó nằm trong bảng kiểm ở mục 12.8.4.

12.6.4 Dựng đồ thị cũng tốn tiền

Còn một chi phí nữa ít ai để ý: chính việc *tạo* node cũng mất thời gian, và thời gian ấy tiêu trong luồng chính chứ không phải luồng âm thanh.

Bảng 12.13 Chi phí tạo và nối node, đo trên bốn cỡ đồ thị

Số node	Thời gian dựng	Mỗi node
64	3,3 ms	26,0 µs
256	11,9 ms	23,3 µs
1.024	47,1 ms	23,0 µs
4.096	227,6 ms	27,8 µs

$$N_{\text{node}}^{\max} = \frac{t_q}{t_{\text{node}}} = \frac{2{,}90\ \text{ms}}{25\ \mu\text{s}} \approx 116 \tag{12.11}$$

Chia ngân sách một khối cho chi phí một node được một con số dễ nhớ: khoảng 116 node. Dựng nhiều hơn thế trong một khung hình là đã tiêu hết thời gian của một khối dựng — và tuy hai luồng khác nhau nên nó không gây ngắt tiếng ngay, nó vẫn làm khung hình giật.

Hệ quả thực hành của con số 116

Đừng dựng đồ thị của cả màn chơi trong một khung hình. Chia ra vài khung, hoặc dựng lúc màn hình đang tải.

Đừng tạo node trong vòng lặp vẽ. Mỗi bước chân một BufferSource là bình thường; ba mươi bước chân trong một khung hình thì không.

Và với những chuỗi hiệu ứng dùng đi dùng lại, hãy dựng sẵn một bể rồi tái sử dụng — chỉ trừ BufferSourceNode, thứ không tái sử dụng được vì đặc tả cấm.

ĐỐI TƯỢNG	VÒNG ĐỜI	DỌN BẰNG CÁCH NÀO
BufferSourceNode	Một lần rồi kết thúc	onended → disconnect() → bỏ tham chiếu
GainNode của giọng	Theo giọng	Dọn cùng source trong chính onended ấy
GainNode của nhóm	Suốt phiên	Không dọn: dựng một lần lúc nạp
ConvolverNode của bus	Suốt phiên	Không dọn (mục 12.5.2)
AudioBuffer đã giải mã	Suốt phiên	Giữ trong Map; chỉ bỏ khi đổi màn chơi
AudioWorkletNode	Theo hiệu ứng	disconnect() và đóng port trước khi bỏ
AudioContext	Suốt phiên	close() khi rời trang; suspend() khi tạm dừng

Hai dòng đỏ trên sinh ra hàng nghìn lần mỗi phút; các dòng còn lại sống lâu hơn một giọng. Nhầm hai nhóm ấy với nhau là gốc của hầu hết lỗi rò.

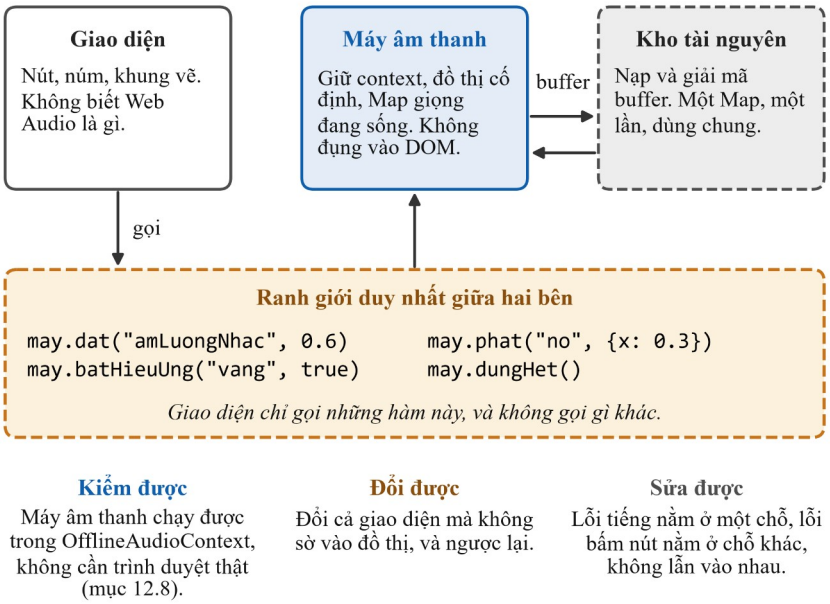
Hình 12.20 Cái gì phải dọn, cái gì phải giữ, và dọn bằng cách nào

12.7 Đóng gói ứng dụng — tách máy âm thanh khỏi giao diện

12.7.1 Ba mô-đun và một ranh giới

Sáu mục vừa rồi cho một đống kỹ thuật. Nếu đổ hết chúng vào cùng một tệp với mã xử lý nút bấm thì sau hai tuần không ai sửa được nữa — kể cả người viết.

Cách tổ chức đã chứng minh được là dùng được chia làm ba mô-đun, và điều quan trọng không phải là con số ba mà là *ranh giới* giữa chúng.



Hình 12.21 Ba mô-đun, và giao diện chỉ được gọi qua một ranh giới duy nhất

Bảng 12.14 Ba mô-đun: việc của mỗi bên, và thứ mỗi bên bị cấm

Mô-đun	Việc của nó	Bị cấm
Giao diện	Đọc trạng thái nút và núm, vẽ đồng hồ mức, gọi các hàm của máy âm thanh	Không được biết tới AudioContext, node, hay currentTime
Máy âm thanh	Giữ context, dựng đồ thị cố định, quản lý giọng sống, lập lịch mọi tham số	Không được đụng vào DOM, không được tự đọc tệp
Kho tài nguyên	Nạp và giải mã buffer, giữ một Map dùng chung, trả buffer khi được hỏi	Không được biết đồ thị mắc thế nào

Một phép thử để biết ranh giới có đúng không

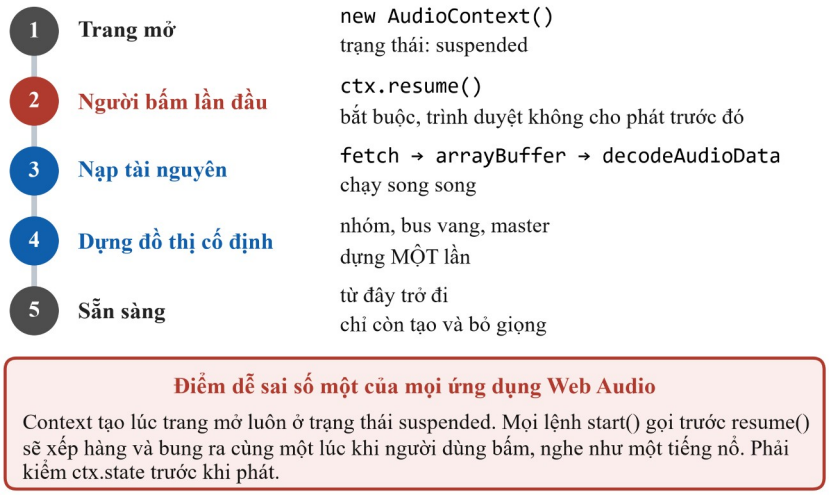
Tìm chữ document và chữ window trong tệp máy âm thanh. Nếu có, ranh giới đã bị phá vỡ.

Tìm chữ createGain, connect, currentTime trong tệp giao diện. Nếu có, ranh giới cũng đã bị phá vỡ.

Phép thử nghiêm nhất: chạy được máy âm thanh trong OfflineAudioContext, không có trình duyệt, không có DOM. Nếu chạy được thì ranh giới đúng — và đó cũng chính là thứ làm cho mục 12.8 khả thi.

12.7.2 Nạp và mở máy — điểm dễ sai số một của mọi ứng dụng Web Audio

Mọi trình duyệt hiện nay đều tạo AudioContext ở trạng thái suspended cho tới khi có một cử chỉ thật của người dùng. Đây là quy định chống quảng cáo tự kêu, và nó không có ngoại lệ.



Nạp tài nguyên và dựng đồ thị có thể đổi thứ tự cho nhau, nhưng mốc người bấm lần đầu thì không: nó do trình duyệt quy định, không do chúng ta.

Hình 12.22 Năm mốc từ lúc trang mở tới lúc ứng dụng sẵn sàng phát

Triệu chứng khi làm sai rất dễ nhận: bấm nút đầu tiên thì im, và tới lần bấm thứ hai thì tất cả những gì đã hẹn từ trước bung ra cùng một lúc thành một tiếng nổ. Nguyên nhân là các lệnh start đã được hẹn vào một đồng hồ chưa chạy.

Liệt kê 12.8 Mở máy đúng cách — resume nằm trong chính trình xử lý sự kiện

```
let may = null;

document.querySelector("#batDau")
  .addEventListener("click", async () => {
    if (!may) {
      may = new MayAmThanh();
      await may.moMay();          // resume() nằm trong này
      await may.napKho(DANH_SACH);
    }
    may.phat("mo_dau");
  }, { once: false });

// Và mọi chỗ phát tiếng đều phải kiểm trước:
phat(ten, opts) {
  if (this.ctx.state !== "running") return null;
  /* ... */
}
```

12.7.3 Khung mã của cả chương gói lại

Toàn bộ những gì đã học trong chương này gói được vào hai mảnh mã. Mảnh thứ nhất dựng một lần và sống suốt phiên; mảnh thứ hai sinh ra hàng nghìn lần và tự dọn.

```
class MayAmThanh {
 async moMay() {
  const c = new AudioContext();
  await c.resume();
  const m = c.createGain();
  const n = c.createGain();
  const r = c.createConvolver();
  n.connect(m); r.connect(m);
  m.connect(c.destination);
  this.ctx = c; this.nhac = n;
  this.vang = r;
  this.song = new Map();
 }
 dat(ten, v, ms = 30) {
  const p = this.param(ten);
  const t = this.ctx.currentTime
   + 0.02;
  p.cancelScheduledValues(t);
  p.setValueAtTime(p.value, t);
  p.linearRampToValueAtTime(
   v, t + ms / 1000);
 }
}
```

(a) Máy âm thanh: hàm dat()

```
phat(ten, muc = 1, gui = 0) {
 const c = this.ctx;
 const b = this.kho.get(ten);
 if (!b) return null;
 const s = c.createBufferSource();
 s.buffer = b;
 const g = c.createGain();
 g.gain.value = muc;
 s.connect(g).connect(this.nhac);
 if (gui > 0) {
  const k = c.createGain();
  k.gain.value = gui;
  g.connect(k).connect(this.vang);
 }
 const id = Symbol();
 s.onended = () => {
  s.disconnect(); g.disconnect();
  this.song.delete(id);
 };
 this.song.set(id, s);
 s.start(c.currentTime + 0.02);
 return id;
}
```

(b) Một giọng: hàm phat()

Hình 12.23 Hai mảnh mã: thứ dựng một lần, và thứ sinh ra hàng nghìn lần

Chú ý ba dòng cancel, neo, ramp trong hàm dat của mảnh (a). Chúng là mục 12.2.4 đóng gói lại thành một chỗ duy nhất, và mọi núm trong ứng dụng đều đi qua đó. Đây là cách duy nhất để không bỏ sót ở lần thứ ba mươi.

Bên phải, hàm phat làm đúng năm việc: lấy buffer từ kho, tạo source và gain, nối nhánh gửi vang nếu có, đăng ký onended để tự dọn, rồi mới start với khoảng an toàn 20 mili giây. Không dòng nào trong hai khối này biết đến DOM.

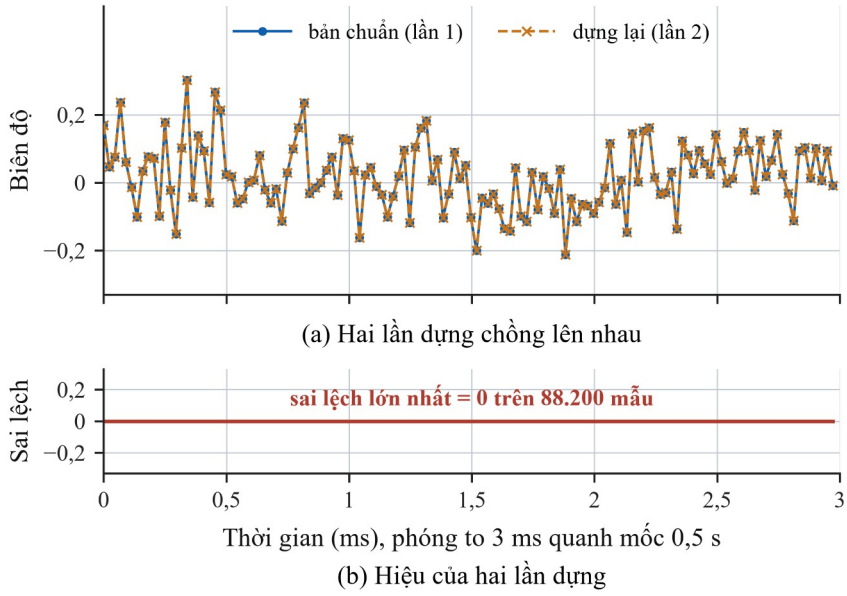
> **Những thứ chưa có trong khung mã trên**
> Xử lý lỗi khi decodeAudioData thất bại — tệp lỗi hoặc định dạng trình duyệt không đọc được.
> Trần đa âm của mục 12.5.3, và bể node tái sử dụng của mục 12.6.4.
> Hàm dungHet() để tắt sạch mọi giọng khi chuyển màn, và close() khi rời trang.
> Ba việc trên là nội dung của bài thực hành 2 cuối chương này.

12.8 Kiểm và đo — làm sao biết lần sửa vừa rồi không làm sai gì

12.8.1 Vì sao kiểm âm thanh khó, và vì sao ở đây nó lại dễ

Kiểm tự động cho phần mềm âm thanh nổi tiếng là khó, vì kết quả phụ thuộc đồng hồ, phụ thuộc tải máy, và phụ thuộc thiết bị ra. Chạy hai lần thì ra hai tệp khác nhau ở những chữ số cuối, và thế là không so sánh được.

OfflineAudioContext gỡ đúng khó khăn ấy. Nó dựng nhanh hơn thời gian thực, không phụ thuộc thiết bị, và — điều quan trọng nhất — nó cho kết quả *lặp lại được*.



(a) Hai lần dựng chồng lên nhau

(b) Hiệu của hai lần dựng

Hình 12.24 Dựng hai lần cùng một đồ thị, và sai lệch giữa hai lần

$$\max_n |y_{\text{mới}}[n] - y_{\text{chuẩn}}[n]| < \varepsilon \qquad (12.12)$$

Phép kiểm hồi quy chỉ là công thức trên: dựng lại, so với tệp chuẩn, và đòi sai lệch nhỏ hơn ε. Điều bất ngờ dễ chịu là ε ở đây lấy bằng đúng *không* được — phép đo cho sai lệch lớn nhất bằng 0 trên 88.200 mẫu. Không phải rất nhỏ, mà bằng không, từng bit một.

Liệt kê 12.9 Một phép kiểm hồi quy đầy đủ, chạy được trong vòng kiểm tự động

```
import { OfflineAudioContext } from "node-web-audio-api";
import { readFileSync, writeFileSync } from "node:fs";

async function dungCanh() {
  const c = new OfflineAudioContext(2, 44100 * 2, 44100);
  const may = new MayAmThanh(c);      // CÙNG mô-đun, khác context
  await may.moMay();
  may.phat("trong", { muc: 0.8, gui: 0.3 });
  may.dat("amLuongNhac", 0.6);
  return c.startRendering();
}

const b = await dungCanh();
const moi = b.getChannelData(0);
const chuan = new Float32Array(readFileSync("chuan.f32").buffer);

let lech = 0;
for (let i = 0; i < moi.length; i++)
  lech = Math.max(lech, Math.abs(moi[i] - chuan[i]));

if (lech > 0) { console.error("LỆCH", lech); process.exit(1); }
```

12.8.2 Ba mức kiểm, từ chặt tới lỏng

Không phải phép kiểm nào cũng nên đòi sai lệch bằng không. Ba mức sau dùng cho ba loại thay đổi khác nhau.

Bảng 12.15 Ba mức kiểm và khi nào dùng mức nào

Mức	Đòi hỏi	Dùng khi	Điểm dễ sai
Từng bit	$\varepsilon = 0$	Sửa thứ đáng lẽ không đổi tiếng: đổi tên biến, tách hàm, dọn mã	Đổi phiên bản trình duyệt cũng làm nó đỏ — và đó là đỏ đúng
Theo ngưỡng	$\varepsilon = 10^{-6}$ tới 10^{-4}	Đổi thứ tự phép cộng, đổi kiểu số, chuyển sang worklet	Ngưỡng quá lỏng thì lỗi thật đi qua
Theo chỉ số	Đỉnh, LUFS, phổ lệch dưới ngưỡng	Đổi thuật toán có chủ ý: đổi bộ lọc, đổi luật trộn	Hai tiếng khác hẳn nhau vẫn có thể cùng LUFS

Một lời khuyên từ kinh nghiệm dựng chính cuốn sách này

Lưu tệp chuẩn dưới dạng float32 thô, không phải WAV nén hay MP3. Mọi phép mã hoá có tổn hao đều làm vô hiệu phép so sánh từng bit.

Lưu kèm một tệp JSON ghi số kênh, tần số lấy mẫu, và phiên bản thư viện đã dùng. Khi phép kiểm đỏ mà không hiểu vì sao, tệp ấy là thứ trả lời.

Và khi phép kiểm đỏ vì chúng ta cố ý đổi thuật toán, đừng nới ngưỡng — hãy *nghe* bản mới, xác nhận nó đúng, rồi thay tệp chuẩn. Nới ngưỡng là cách chắc chắn nhất để phép kiểm mất giá trị.

12.8.3 Đo trong lúc chạy thật

Kiểm ngoại tuyến bắt được lỗi về nội dung. Nó không bắt được lỗi về tài nguyên, vì nó không chạy trong thời gian thực. Ba số sau nên đọc ngay trong ứng dụng đang chạy.

- **ctx.renderCapacity —** phần trăm ngân sách một khối đã dùng. Vượt 70 phần trăm thường xuyên là dấu hiệu phải giảm tải trước khi máy yếu hơn gặp phải. Đã bàn ở mục 11.5.3.
- **Kích thước Map giọng đang sống —** nó phải dao động quanh số giọng thật và phải *trở về rỗng* khi im tiếng. Không về rỗng là rò, và đây là phép phát hiện rẻ nhất.
- **Số node đã tạo từ lúc mở —** một biến đếm tự tăng. Chia cho thời gian chạy được số node mỗi giây, và so với con số 116 node mỗi khối ở công thức (12.11).

Bảng 12.16 Ba số đo trong lúc chạy, ngưỡng, và việc phải làm khi vượt

Số đo	Đọc từ đâu	Ngưỡng đáng lo	Việc phải làm
Tải luồng âm thanh	ctx.renderCapacity	Trên 70 phần trăm kéo dài	Giảm tải theo thứ tự ở bảng mục 12.5.4
Số giọng đang sống	Kích thước Map của máy âm thanh	Không về 0 sau khi im tiếng	Tìm chỗ thiếu onended — mục 12.6.2
Số node tạo mỗi giây	Một biến đếm tự tăng	Trên 116 node trong một khung hình	Dựng sẵn và tái sử dụng — mục 12.6.4

Ba số này nên hiện ngay trên màn hình trong bản dựng dành cho lập trình viên, cạnh số khung hình mỗi giây. Chúng rẻ tới mức không có lý do gì để không đo, và cả ba đều phát hiện được lỗi trước khi người dùng nghe thấy.

12.8.4 Bảng kiểm trước khi giao



Hình 12.25 Tám dòng phải qua trước khi giao một ứng dụng Web Audio

Bảy trong tám dòng của bảng đã có phép đo kèm theo ngay trong chương này. Dòng còn lại — mở máy đúng cách — kiểm bằng cách tải trang trong một cửa sổ mới và bấm nút đầu tiên: nếu tiếng ra ngay và ra đúng một lần, dòng ấy qua.

Bảng 12.17 Kiểm mỗi dòng của bảng bằng cách nào, và con số để so

Dòng	Kiểm bằng cách nào	Con số để so
Không kêu tách	Dựng ngoại tuyến một lần kéo núm, đo năng lượng trên 5 kHz	Nên dưới −110 dB; xem mục 12.2.2
Huỷ đúng chỗ	Cắt ngang một ramp dài bằng một ramp ngắn, đọc giá trị cuối	Phải dừng ở đích mới, không bị kéo về đích cũ
Bypass rẻ thật	Dựng cùng một cảnh ba lần: bật, gain 0, ngắt nối	Bản ngắt nối phải rẻ hơn rõ rệt — bài này đo 3,5 lần
Vang dùng chung	Đếm số ConvolverNode trong đồ thị	Phải bằng số phòng, không phải số giọng
Có trần đa âm	Phát dồn dập rồi đọc kích thước Map giọng sống	Không được vượt trần đã đặt
Không rò node	Chờ im tiếng rồi đọc lại kích thước Map ấy	Phải bằng 0
Mở máy đúng cách	Tải trang trong cửa sổ mới, bấm nút đầu tiên	Tiếng ra ngay, và ra đúng một lần
Có bản chuẩn	Chạy phép kiểm ở mục 12.8.1 hai lần liên tiếp	Sai lệch bằng 0 trên toàn bộ số mẫu

12.9 Thực hành, bài tập và tổng kết

12.9.1 Ba bài thực hành

Ba bài dưới đây làm ngay sau khi đọc xong chương, mỗi bài khoảng mười lăm phút, và cả ba dùng chung một tệp khởi đầu.

- **Thực hành 1 — chữa một thanh trượt.** Cho sẵn một trang có thanh trượt nối với gain bằng phép gán thẳng. Nghe tiếng rè, sửa theo mục 12.2.2, rồi dựng ngoại tuyến cả hai bản và đo phần năng lượng trên 5 kilohertz của mỗi bản. Ghi lại hai con số.
- **Thực hành 2 — thêm ba thứ còn thiếu vào khung mã.** Lấy khung mã ở mục 12.7.3, thêm trần đa âm, thêm hàm dungHet, và thêm xử lý lỗi cho decodeAudioData. Kiểm bằng cách phát ba mươi tiếng liên tiếp rồi in kích thước Map giọng sống.
- **Thực hành 3 — dựng một phép kiểm hồi quy.** Chạy mã ở mục 12.8.1 để sinh tệp chuẩn, sửa một hằng số bất kỳ trong máy âm thanh, chạy lại, và xác nhận phép kiểm đỏ. Rồi hoàn nguyên và xác nhận nó xanh.

12.9.2 Bảy bài tập

- **Bài 1 — đo lại tỉ số bus vang trên máy của người đọc.** Chạy phép đo của mục 12.5.2 với bốn số giọng, vẽ hai đường và so hình dạng với hình trong bài. Con số sẽ khác; hình dạng phải giống. Giải thích một câu nếu không giống.
- **Bài 2 — bypass ba cách.** Dựng cùng một rack ba lần: bypass bằng gain 0, bằng disconnect thẳng, và bằng mờ chéo rồi disconnect. Đo cả thời gian dựng lẫn năng lượng cao tần quanh chỗ chuyển. Lập một bảng ba dòng hai cột.
- **Bài 3 — kiểm cancelAndHoldAtTime.** Trên trình duyệt của người đọc, dựng phép thử của mục 12.2.4 với ba biến thể: không huỷ, huỷ rồi neo, và cancelAndHoldAtTime đứng riêng. Nói xem trình duyệt ấy có cài đúng đặc tả không.
- **Bài 4 — luật wet/dry riêng.** Cài một luật thứ ba nằm giữa tuyến tính và đẳng công suất, ví dụ luật bù một nửa. Đo độ gợn trên toàn dải núm và so với hai con số 2,99 và 0,03 trong bài. Nói khi nào luật ấy đáng dùng.
- **Bài 5 — ducking theo đường bao.** Thay ducking theo sự kiện bằng ducking đọc mức từ AnalyserNode trên nhóm lời. Đo lại chênh lệch lời trên nhạc và so với con số 8,43 decibel trong bài. Nói xem cách nào chính xác hơn và vì sao.
- **Bài 6 — tìm rò trong một ứng dụng cho sẵn.** Cho một trang có ba chỗ rò node khác nhau. Tìm cả ba bằng cách đếm Map giọng sống, sửa, và chứng minh bằng một phép kiểm tự động rằng Map trở về rỗng sau khi im tiếng.
- **Bài 7 — bảng kiểm cho dự án tự làm. Áp bảng kiểm tám dòng ở mục 12.8.4 vào dự án thực hành của chính người đọc. Với mỗi dòng, ghi rõ đã qua hay chưa, và nếu chưa thì ghi việc cần làm. K**èm ít nhất hai phép đo tự chạy.

12.9.3 Tổng kết chương 12



Hình 12.26 Chương 12 gói lại trong sáu con số

Chương 11 và chương 12 hợp thành phần Web Audio của cuốn sách, và hai chương ấy trả lời hai câu hỏi khác nhau. Chương 11 hỏi: một đồ thị âm thanh thời gian thực hoạt động như thế nào. Chương 12 hỏi: làm sao để nó chịu được người dùng thật, máy thật, và một người khác đến sửa mã.

> **Ba điều đáng mang sang chương sau**
>
> Mọi thứ đổi trên đường tín hiệu đều phải đổi có hình dạng — không bao giờ đổi bằng một bước nhảy. Tai người phát hiện bước nhảy giỏi hơn phát hiện bất cứ thứ gì khác.
>
> Tối ưu là làm *ít đi*, không phải làm nhanh hơn. Sáu việc trong bảng ở mục 12.5.4 đều là bỏ bớt việc chứ không phải tính nhanh hơn — và tỉ số của chúng đi từ 1,05 tới 16,2 lần.
>
> Và thứ đắt nhất trong một đồ thị Web Audio không phải là phép tính, mà là những node bị bỏ sót.

Chương 13 rẽ sang một hướng khác hẳn: mô hình AI tạo sinh nhạc và giọng — biểu diễn bằng token, họ mô hình tự hồi quy và khuếch tán, và bài toán tách nguồn. Nhưng mọi thứ dựng ra ở đó rồi cũng phải chạy trong một đồ thị như đồ thị của chương này, và phải chịu đúng những ràng buộc vừa đo.

Tài liệu tham khảo của chương

Nguồn trực tiếp của nội dung trong bài

- W3C — Web Audio API, khuyến nghị chính thức. Phần AudioParam automation là nguồn trực tiếp của mục 12.2: ngữ nghĩa chính xác của từng phương thức lập lịch, và định nghĩa của cancelScheduledValues với cancelAndHoldAtTime. Đọc kỹ phần này trước khi tin vào trực giác.
- Rothstein, J. — MIDI: A Comprehensive Introduction, chương về quản lý đa âm. Khái niệm trần đa âm và luật chọn giọng để cắt ở mục 12.5.3 lấy từ truyền thống nhạc cụ điện tử, không phải từ Web Audio.

- Izhaki, R. — Mixing Audio: Concepts, Practices and Tools. Nguồn của mục 12.3.5 về thứ tự khe và của mục 12.4 về bus, nhóm, và ducking. Sách viết cho người trộn nhạc chứ không cho người lập trình, và đó là lý do nên đọc.
- Choi, H. — Web Audio performance and best practices, cùng loạt bài về kiến trúc luồng. Nguồn của phần bàn về chi phí dựng đồ thị ở mục 12.6.4.

Sách

- Smus, B. — Web Audio API. Phần về bể node tái sử dụng và về quản lý giọng vẫn còn nguyên giá trị.
- Reiss, J. D., McPherson, A. — Audio Effects: Theory, Implementation and Application. Chương về hiệu ứng thời gian giải thích vì sao vang phải nằm cuối chuỗi, và chương về động lực giải thích thứ tự nén với EQ.
- Somberg, G. (chủ biên) — Game Audio Programming: Principles and Practices. Bộ sách nhiều tập, và các chương về voice management, ducking, và bus routing là bản dài của mục 12.4 và 12.5.
- Beazley, D. — Python Distilled, chương về vòng đời đối tượng. Đọc để đối chiếu: bài toán ở mục 12.6 không phải bài toán riêng của Web Audio.

Công cụ nên biết

- Thẻ Performance của bảng điều khiển trình duyệt. Nó vẽ riêng một hàng cho luồng âm thanh, và đây là cách nhìn thấy chi phí dựng đồ thị ở mục 12.6.4 rơi vào luồng nào.
- Thuộc tính ctx.renderCapacity. Con số duy nhất đọc được ngay trong mã cho biết ứng dụng còn bao nhiêu chỗ thở. Đã bàn ở mục 11.5.3 và dùng lại ở mục 12.8.3.
- node-web-audio-api. Bản cài đặt Web Audio đầy đủ chạy ngoài trình duyệt; toàn bộ số liệu của chương này dựng bằng nó, và nó là thứ làm cho phép kiểm hồi quy ở mục 12.8 chạy được trong vòng kiểm tự động.
- Bất kỳ thư viện đo LUFS nào theo ITU-R BS.1770. Hai chỉ số đỉnh và độ to ở mục 12.3.5 là hai chỉ số tối thiểu để nói chuyện về thứ tự khe mà không cãi nhau.

Đọc thêm theo hướng

- **Muốn đi sâu về quản lý giọng:** tìm đọc về voice stealing và về priority-based voice allocation trong các máy tổng hợp phần cứng. Trần đa âm ở mục 12.5.3 mới là bản đơn giản nhất của cả một họ thuật toán.
- **Muốn đi sâu về trộn động:** tìm đọc về adaptive music và về vertical remixing trong âm thanh trò chơi. Ducking ở mục 12.4 là trường hợp riêng đơn giản nhất của việc bản trộn tự đổi theo trạng thái.
- **Muốn đi sâu về kiểm:** tìm đọc về golden file testing và về perceptual audio metrics như PEAQ hay ViSQOL. Ba mức kiểm ở mục 12.8.2 dừng ở chỗ so sánh số; các chỉ số ấy đi tiếp tới chỗ so sánh nghe.

– **Muốn đi sâu về kiến trúc:** tìm đọc mã nguồn của Howler.js và của Tone.js. Cả hai đều là bản đầy đủ của mục 12.7, và đọc chúng là cách nhanh nhất để thấy những gì khung mã trong bài còn thiếu.

Một lưu ý về các con số trong bài

> **Cách đọc các con số**
>
> Các con số về decibel — nhiễu răng cưa, tiếng tách khi bypass, độ hõm của luật wet/dry, đỉnh và độ to khi đổi thứ tự khe, chênh lệch lời trên nhạc — là kết quả *đo* trên tiếng ra thật của đồ thị Web Audio chạy thật, đo lại bằng mô-đun Python của các chương trước. Chúng lặp lại được trên mọi máy.
>
> Các con số về *thời gian dựng* — mọi tỉ số trong mục 12.5 và 12.6 — là số đo trên máy dựng cuốn sách này. Con số tuyệt đối sẽ khác trên máy khác; *tỉ số* và *hình dạng* của đường cong thì không.
>
> Con số 116 node ở công thức (12.11) là phép chia giữa ngân sách một khối và chi phí trung bình một node đo được. Nó là bậc độ lớn để nhớ, không phải một hằng số của đặc tả.
>
> Sai lệch bằng 0 ở mục 12.8.1 là con số chính xác, không phải làm tròn: 88.200 mẫu trùng nhau từng bit qua hai lần dựng trong hai tiến trình khác nhau.
>
> Mã JavaScript in trong bài là mã đã chạy, không phải mã giả.