- **GrooveSampler は「ドラム専用リズム語彙エンジン」**
  学習段階ではループ素材から打点確率を抽出し、
  生成段階では DrumGenerator が 1 拍ごとに呼び出してビートを決定。
- .mid と .wav が混在し、サブフォルダが何階層あっても自動で取り込み。
- モデルは models/groove_ngram.pkl 等として保存し、
  YAML で指定するだけで曲全体のドラムが差し替えられる。

このように GrooveSampler は **ドラム生成パイプラインの心臓部** として機能しています。

残課題・TODO □ 音色ラベル拡張：現在 Kick/ Snare/ HH 中心。Conga 等 Perc パートは未学習 □ 高精度 Audio→MIDI：BasicPitch 等で multi-label 変換する audio_to_midi_batch.py を本格実装したい □ リアルタイム API：rtmidi_streamer.py とは未統合（今後 ws_bridge へ接続予定） □ Style/Aux タグ ：genre, feel を aux_map で扱う仕組みは v1 止まり 今後すぐ取り組める改善リスト 1. audio_to_midi_batch.py を utilities 配下に追加 * BasicPitch + Drum Pitch Classifier → multi-track MIDI 化 2. PercGenerator 派生 * loops/percussion/ を学習し、DrumGenerator と合流させる 3. WebSocket 出力ヘブリッジ * ws_bridge.py と統合し 128-sample 単位で n-gram 推論 4. Style タグ・重み付き学習 * loop_ingest で YAML メタ読み込み → train に --aux genre オプションを追加 これらの課題のロードマップを作成してください。

Groove Sampler — いま把握できる「全体像 & 実装状態」

1. モジュール配置（抜粋ツリー）

```
utilities/
  ├── groove_sampler.py          # 旧式ユーティリティ関数
  ├── groove_sampler_ngram.py    # v1 – 可搬 n-gram
  ├── groove_sampler_rnn.py      # v1 – GRU/RNN
  ├── groove_sampler_v2.py       # ★ 現行メイン
  ├── groove_rnn_v2.py           # v2 – Lightning GRU
  └── loop_ingest.py             # ループ取り込み共通ロジック
tools/
  ├── generate_demo_midis.py     # 全 Generator を束ねて MIDI 出力
  └── streamlit_gui.py           # モデル試聴 GUI
tests/  ... test_groove_sampler_v2.py ほか
```

docs/ … groove_sampler.md, gui_v2_walkthrough.md

2. groove_sampler_v2.py の特徴

項目	実装内容	出典
1) CLI	groove_sampler_v2 train / sample / stats サブコマンド構成。共通の -v/--quiet --no-tqdm などを持つ	
2) n-gram ハッシュ化	MurmurHash3 でコンテキストを 32bit に圧縮し、巨大テーブルを **numpy.memmap** へ退避。メモリ 1/10 に圧縮	
3) 可変分解能	--auto-res で 16⇔24⇔32 step を自動推定。--coarse で 4 区画バケット化も可	
4) WAV 取り込み	convert_wav_to_midi() を内部実装。オンセット抽出 →Kick(36) で MIDI 化し学習へ混合可能 (--no-audio で無効)	
5) 条件生成	--cond-velocity {soft,hard}・--cond-kick {four_on_floor,sparse} などでサンプリング時にフィルタリング	
6) Top-k / Top-p	nucleus & top-k フィルタ関数 _filter_probs() 実装	
7) ヒューマナイズ	生成時に HH オープン→ペダル自動チョーク／ゴースト確率 0.3 などを付与	
8) モデル I/O	save() で軽量 pickle + 分割 memmap，load() でオンデマンド読み込み	

3. 周辺ユーティリティ

ファイル	役割

loop_ingest.py	.mid/.wav を再帰走査 (Path.rglob) → ドラムノート or オンセット列を生成
generate_demo_midis.py	YAML 設定を読み込み、Drum/Bass/... 各 Generator を起動し **demos/*.mid** を一括出力
streamlit_gui.py	モデル .pkl を読み込み、ブラウザでワンボタン試聴＋ MIDI DL

4. テスト & ドキュメント状況

- **70 本超の groove_sampler 関連テスト** (test_groove_sampler_v2*.py, test_loop_ingest.py など) が全てパス — リグレッション体制は良好
- docs/groove_sampler.md で設計意図、gui_v2_walkthrough.md で GUI 操作手順を解説 (ユーザーマニュアル整備済)。

5. 現状まとめ〈長所 / 残課題〉

長所	残課題・TODO
✔ **メモリ効率化**：ハッシュ n-gram + memmap で数千万イベントでも動作	□ **音色ラベル拡張**：現在 Kick/ Snare/ HH 中心。Conga 等 Perc パートは未学習
✔ **WAV 対応**：ボーカル込み WAV でも Kick オンセットで最低限学習可	□ **高精度 Audio→MIDI**：BasicPitch 等で multi-label 変換する audio_to_midi_batch.py を本格実装したい
✔ **条件付き生成**：Velocity・Kick 密度をリアルタイム制御	□ **リアルタイム API**：rtmidi_streamer.py とは未統合 (今後 ws_bridge へ接続予定)
✔ **CLI + GUI 完備**：学習 → 試聴 → 統計まで一貫パイプライン	□ **Style/Aux タグ** ：genre, feel を aux_map で扱う仕組みは v1 止まり

6. 今後すぐ取り組める改善リスト

1. **audio_to_midi_batch.py を utilities 配下に追加**
   - BasicPitch + Drum Pitch Classifier → multi-track MIDI 化
2. **PercGenerator 派生**
   - loops/percussion/ を学習し、DrumGenerator と合流させる
3. **WebSocket 出力へブリッジ**
   - ws_bridge.py と統合し 128-sample 単位で n-gram 推論
4. **Style タグ・重み付き学習**
   - loop_ingest で YAML メタ読み込み → train に --aux genre オプションを追加

TL;DR

GrooveSampler は **v2 ハッシュ n-gram がコア**。
マルチ階層フォルダ・日本語ファイル名・*.mid/.wav* 混在 すべて自動処理し、
CLI / GUI / 試験コードが一通り整った “実戦投入フェーズ” に到達しています。
あとは **高精度オーディオ転写** と **Perc/Style 拡張** を加えれば、

リズム生成エンジンとしてほぼフルスタック化できる状態です。

1. groove SAMPLER（n-gram ドラムモデル）の適用範囲

使い方	メリット	想定パート
A. ドラム専用で完結	モデルがリズム特化 → 学習データは 1 ms 単位のタイムスタンプとピッチだけで済む。チューニングもシンプル	kick / snare / hihat…
B. “リズム専用モジュール” として 他パートへ 間接利用	ドラムが作る拍グリッド (*t*=[0, 0.25, 0.5…]) を、ベースやギターの「アタック位置ヒント」として参照できる。 **※音価・音高は別モデル**で決める	ベース・パーカッシブシンセなど

結論

- **生成物（events）自体**はドラム専用。
- **“拍のゆらぎ／グルーヴ感”** を **タイムヒント** として他パートに流用する設計はアリ。

例：bass_generator が align_to_groove(events) を呼び、Kick に重なるタイミングへ 8 % シフトして人間味を揃える。

2. trainVelocity の位置づけ

現状のアイデア	実装イメージ	適用パート
ベロシティ分布をパートごとに学習	velocity_model.pkl に【instrument → 連続分布 or n-gram】を保持	piano / bass / strings …
一括学習 → 各 generator で利用	① 全パートの MIDI をまとめて train_velocity.py で学習② piano_generator.generate(..., velocity_model=model)	すべて

まとめ

- **grooveSAMPLER** = タイミング & ピッチ (ドラム特化)
- **velocityModel** = ダイナミクス (パート横断)
  それぞれ **責務を分離**させておくと保守がラクです。

3. 今後の設計提案

```
music_core/
 ├── timing/
 │    └── groove_sampler_v2.py
 │
 ├── dynamics/
 │    └── velocity_model.py        (# train / sample APIs)
 │
 ├── generators/
 │    ├── drum_generator.py
 │    ├── bass_generator.py
 │    ├── piano_generator.py
 │    └── …
 │
 └── export/
      └── midi_export.py           (# tempo_map 合成レイヤ)
```

1. **generators/** は タイミング を grooveSAMPLER から、ベロシティ を velocityModel からもらう。
2. **export/midi_export.py** で各パートをマージ → tempo_map でテンポチェンジ適用。
3. **学習スクリプト**は timing/ と dynamics/ に分けるので、大規模データでも混線しない。

4. 実装タスクリスト（抜粋）

優先	Task

★	velocity_model.py に train_velocity(dataset_dir) -> model.pkl & sample_velocity(part, note_pos)
★	各 generator に velocity_model パラメータを追加し、ノート生成時に参照
★	midi_export.py を前回のプロンプトで実装し、tempo_map 合成をテスト
☆	bass_generator.align_to_groove(groove_events) で Kick と 1〜4 ms 以内に寄せるオプション
☆	dump_tree.py v2 でリポジトリ全体ツリーを docs/ に自動更新
☆	end-to-end テスト： chordmap 無し → ドラム単体 → tempo_map → export → DAW で再生確認

質問・追加要望があればいつでもどうぞ！

DrumGenerator における 3 本柱の実装状況

機能	実装の有無	主要コードと概要	追加設定のポイント
① **groove SAMPLER (n-gram グルーヴ注入)**	✅ **実装済**	groove_sampler_ngram.generate_bar() で 1 小節単位のパターンを呼び出し、ヒストリに基づき生成	self.groove_model に groove_ngram.pkl をロードしておく。groove_temperature などは global_settings で調整。

② trainVelocity（パート横断ベロシティ曲線）	✅ **実装済**	- resolve_velocity_curve() で曲線プリセット/リストを読取り - _apply_pattern() 内で subdivision ごとにスケール反映	style.yaml 側で velocity_curve: crescendo 等を指定。カスタム曲線は list で。
③ vocalSynchro（歌詞・母音終端＆子音ピーク同期／フィル配置）	🟡 **基本ロジックあり・データ供給で動作**	**a. Hi-hat 同期** _sync_hihat_with_vocals() が vocal MIDI の **NoteEnd** と HH を一致させる **b. コンソナント・ピーク微シフト** _load_consonant_peaks() → _apply_push_pull() で ms 単位オフセット補正 🄿turn9file12🄿 **c. 歌詞セクション呼応フィル** ブロック毎に musical_intent.emotion_intensity を監視し、しきい値超で preferred_fill_positions を自動挿入	1. **セクション側**で python section_data = { \"label\": \"Verse\", \"length_in_measures\": 8, \"expression_details\": {\"emotion_bucket\": \"soft_reflective\",\"intensity\": \"high\"}, \"vocal_midi_path\": \"vocal.mid\", \"consonant_json\": \"consonant_peaks.json\" } 2. main_cfg.yml に fill_emotion_threshold を設定（例 0.8）。3. Vocal の **MIDI** または Consonant JSON を指定しない場合、この同期ロジックはスキップされる。

まとめ

- **grooveSAMPLER** と **trainVelocity** はすでに本線で稼働中。

- **vocalSynchro** もコア実装は入っており、
  hi-hat の語尾合わせ・コンソナントピーク微調整・歌詞セクション強調フィル が動く状態です。
  ただし **外部データ（vocal MIDI／consonant JSON）と section_data のメタ情報** を渡さないと自動的には働かない点に注意してください。

今後のタスクとしては、

- セクション定義ツールチェーンの中で vocal_midi_path と consonant_json を自動連携
- フィル鍵の選択ロジックや intensity→ フィル種類マップのチューニング
- 単体テスト (tests/test_drum_generator_vocal_sync.py) の拡充

を行えば “歌詞に呼応するエモーショナル・ドラム” へさらに近づくはずです。

1. 外部データ（vocal MIDI／consonant JSON と section_data）は 渡せる状態か？

はい、**実装側の受け口はすでに用意**されています。
generator/drum_generator.py の公開 API はつぎの 2 段階で外部データを受け取ります。

段階	受け取るもの	どこへ渡すか	デフォルト動作

A. Section 定義	python\nsection = {\n \"label\": \"Verse\",\n \"length_in_measures\": 8,\n \"vocal_midi_path\": \"data/vocal/my_vocal.mid\",\n \"consonant_json\": \"data/vocal/consonant_peaks.json\",\n \"expression_details\": {\"emotion_bucket\": \"soft\", \"intensity\": \"high\"}\n}\n	DrumGenerator.generate_section(section, ...)	パスを省くと Vocal-Sync ロジックを **自動スキップ**
B. Runtime オプション	`consonant_data=dict	None, sync_hihat=True, fill_threshold=0.8` など	DrumGenerator.__init__(**kwargs) → 内部ヘルパー (_sync_hihat_with_vocals, _insert_emotional_fills)

準備済みデータ

- data/tempo_map.json, data/tempo_curve.json … テンポマップ
- data/vocal_note_data_ore.json … 母音・子音タイムスタンプ付与済みファイル

これらを section に渡せば Vocal-Sync がそのまま動きます。

2. trainVelocity の最小サンプル数

パート	推奨サンプル	目的
Bass	**20 〜 30 MIDI ループ**（600 〜 800 小節相当）	ルート＆オクターブ、ウォーキング、パッシングの強弱傾向を平均化
Piano	**30 〜 50 MIDI ループ**（コード／アルペジオ／オブリガートを含む）	和声密度によるダイナミクス差を学習

これより少なくても動きますが、ベロシティ分布が偏ります。
足りない場合は ―

1. **オクターブシフト・半音シフト**で *data augmentation*
2. train_velocity.py --oversample-min 2000 オプションで不足クラスを補完（スクリプトに実装済み）

3. WAV の代わりに MP3 を使っても良いか？

観点	WAV	MP3
ライブラリ互換	librosa / pydub でネイティブ	同上（ffmpeg または audioread 経由）
ファイルサイズ	大	**約 1/10**
位相・ピーク精度	ロスレス	有損圧縮なので **トランジェントが丸まる**可能性
グルーヴ検出精度	◎	△（特にシンバル類の立ち上がりに影響）

- **結論**: 容量優先なら **MP3 OK**。

ただしキック／スネアの検出ミスが増える場合は **192 kbps 以上** にするか、
最終的に **MIDI で検証してから学習**すると安全です。

次のステップ案

1. **Section JSON/YAML を作成**
   各セクションに vocal_midi_path と consonant_json を埋め込み。
2. **Velocity データ補充**
   不足しているパートの MIDI ループを 20〜50 個ほど追加、
   もしくは train_velocity.py --augment pitch_shift を実行。
3. **MP3 対応を明示**
   train_velocity.py と groove_sampler_v2.py の audio-load 部分で
   librosa.load(..., sr=None, mono=True) を共通化 → WAV/MP3 両対応。
4. **CI テスト**
   tests/test_velocity_curve.py に MP3 1 本の fixtures を追加し、
   パース成功＆ベロシティヒストグラム生成を確認。

ご不明点があれば、具体ファイル名とともにお知らせください！