

当一个民谣小哥抱着吉他哼唱着《情非得已》时，他右手扫着音孔处的琴弦，左手变换着按着琴颈处的琴弦，一段简单的弹唱便看起来有模有样。在不看脸不看唱功的情况下，是什么原理才赋予这位小哥如此风骚的魅力呢？

这就是吉他伴奏。

而他只是一个吉他初学者，还没办法给歌曲编配伴奏，只好从网上找来吉他谱，按照里面的标识来进行弹奏。他找到了下面这样的谱子：

和弦指法 {

六线谱 {

简谱 {

歌词 {

5 3 5 1 1 - | 1 - - ) 3 4 | 5 4 4 5 4 3 2 | 2 - - 3 2 | 1 1 i i 3 5 |

难以 忘记 初 次 见 你 一 双 迷 人 的 眼 睛

Em F Em F G

5 - - 1 1 | 2 1 1 1 0 i | 7 6 5 5 - | 1 - 1 1 3 1 2 | 2 - 0 3 3 4 |

在 我 脑 海 里 你 的 身 影 挥 散 不 去 握 你 的

这是一个典型的吉他弹唱谱该有的样子，它可以被分成四个部分：

- 和弦指法
- 六线谱
- 简谱
- 歌词

对于初学者，吉他入门的坎儿在于左手的指法，当时我记下了大多数和弦的指法图，左手指尖的磨出的茧也是起了褪，褪了起，身为乐理渣的我终有一天疑惑了，问号三连：

1. 这个和弦为什么叫这个名字？
2. 这个和弦为什么是这个指法？
3. 同一和弦在吉他上到底有多少种不同的指法？

本文将基于基本的乐理知识，用代码推导计算出以上问题的答案，并将其结果可视化。

## 一、从一个单音说起

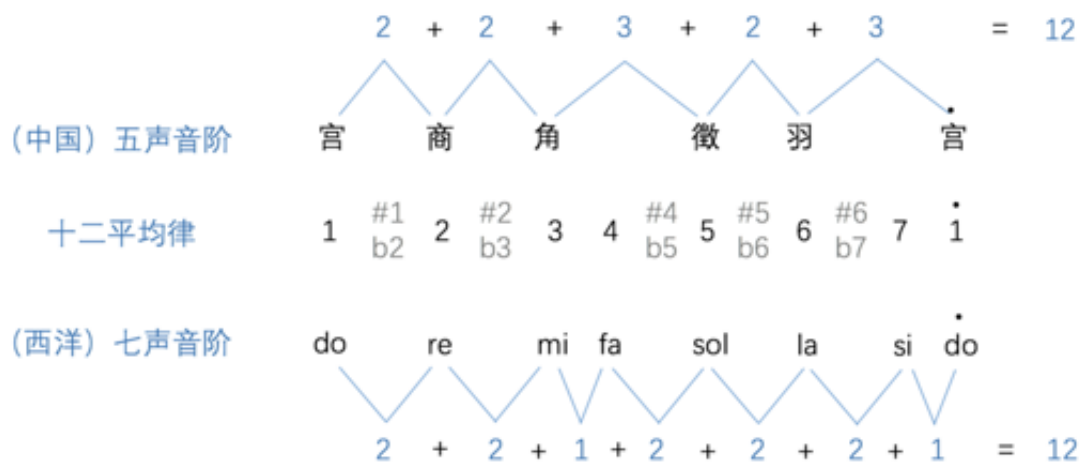
心虚的声明：外行人基于自己的理解强行解释乐理，望专业人士轻喷

声音因物体振动而产生，每一个不同频率（即不同音高）的声响都可以称之为一个单音，但人耳的辨音能力有限，故将人耳能清晰分辨的最小的音高间隔称为 半音 ；

相隔半音的两个音的频率比值为 2 的 12 次方根 。

为什么是这个值，这就得提到 十二平均律 。

音乐界老前辈经过大量的听力实践后，发现例如 do 到 高音do 这个音程作为一个循环听起来最和谐，并且这 高音do 与 do 的频率比率刚好是 2 ，在保证单音之间跨度和谐、而且能较清晰地辨听的情况下，将这个音程按频率比划分成了 12 等份 ，这与中国的五声音阶（宫商角徵羽）和西洋的七声音阶存在相互映照的关系，如下图（这里我暂时用数字标记十二平均律音程上的每个音）：



类似 do 与 高音do 之间的关系在七声音阶里被称为 八度

；

也就是说一个音与它对应高八度的音之间的跨度便是一个音程，它们的频率比为 1: 2 。

1 (do) 与 2 (re) 之间是一个 全音 的跨度，而 3 (mi) 与 4 (fa) 、 7 (si) 与 1. (高音do) 之间是一个 半音 的跨度，一个全音跨度就相当于两个半音跨度，可以看出 1 (do) 与 2 (re) 之间还夹了一个音，我们称它为 #1 (升do) 或者说 b2 (降re) 。

理解了这些后，便可以用代码实现一个单音类：

## 1. 首先来确定一种单音的书写形式

可以借用简谱的标记方式，数字 1、2、3、4、5、6、7 ，分别代表唱名的 do、re、mi、fa、sol、la、si ；

当这个音 升半调 时，在数字的前面加上 # ，例如 #1（升do） ， 降半调 时，在数字前面加上 b ，例如 b1（降do） ；

当标记一个音的高八度音时，在数字的右侧加一个“点号”，例如 1.（高音do） ， #2.（高音升re） （因为字符串没法像简谱那样在数字顶部加点号），当标记一个音的低八度音时，在数字的左侧加一个“点号”，例如 .1（低音do） ， .b2（低音降re） ；

## 2. 构建单音类

```
function is(data) {
  return function(type) {
    return Object.prototype.toString.call(data) === `[object ${type}]`;
  }
}

// 单音类，用于音的映射查询与音高的改变，同时可标记记录其在吉他上的位置
class Tone {
  constructor(toneString = '1', string, fret) {
    // 所有唱名数组
    this.syllableMap = ['do', 're', 'mi', 'fa', 'sol', 'la', 'si'];
    // 音程
    this.keyMap = ['1', ['#1', 'b2'], '2', ['#2', 'b3'], '3', '4', ['#4', 'b5'], '5', ['#5', 'b6'], '6', ['#6', 'b7'], '7'];
    // 所有调名
    this.intervalMap = ['C', ['#C', 'bD'], 'D', ['#D', 'bE'], 'E', 'F', ['#F', 'bG'], 'G', ['#G', 'bA'], 'A', ['#A', 'bB'], 'B'];
    // 单音的字符串表示
    this.toneString = toneString;
    // 单音的字符串表示（去除八度标记）
    this.toneNormal = toneString.replace(/\.\/g, '');
    // 数字音
    this.key = toneString.replace(/\.|b|#/g, '');
    // 唱名
    this.syllableName = this.syllableMap[+this.key - 1];
    // 降半调标记
    this.flat = toneString.match('b') ? 'b' : '';
    // 升半调标记
    this.sharp = toneString.match('#') ? '#' : '';
    let octave_arr = toneString.split(this.key);
    let octave_flat = octave_arr[0].toString().match(/\.\/g);
    let octave_sharp = octave_arr[1].toString().match(/\.\/g);
    // 八度度数
    this.octave = (octave_sharp ? octave_sharp.length : 0) - (octave_flat ? octave_flat.length : 0);
    // 吉他按弦位置
    this.position = {
```

```

        // 第几弦
        string: string,
        // 第几品格
        fret: fret
    };
}
// 获取某个音在音程上的位置
findKeyIndex(keyString) {
    return this.keyMap.findIndex((item) => {
        if (is(item)('Array')) {
            return item.includes(keyString);
        } else if (item === keyString) {
            return true;
        } else {
            return false;
        }
    });
}
// 音高增减, num为增或减的半音数量
step(num) {
    let keyString = this.flat + this.sharp + this.key;
    let len = this.keyMap.length;
    let index = this.findKeyIndex(keyString);
    if (index > -1) {
        num = +num;
        // 计算改变音高后的音在音程上的位置
        let nextIndex = parseInt(index + num, 0);
        let octave = this.octave;
        if (nextIndex >= len) {
            let index_gap = nextIndex - len;
            octave += Math.floor(index_gap / len) + 1;
            nextIndex = index_gap % len;
        } else if (nextIndex < 0) {
            let index_gap = nextIndex;
            octave += Math.floor(index_gap / len);
            nextIndex = index_gap % len + len;
        }
        let nextKey = this.keyMap[nextIndex];
        // 计算并添加高低八度的记号
        let octaveString = new Array(Math.abs(octave)).fill('.').join('');
    );
    let toneString = '';
    if (!is(nextKey)('Array')) {
        toneString = (octave < 0 ? octaveString : '') + nextKey + (octave > 0 ? octaveString : '');
        return new this.constructor(toneString, this.position.string, this.position.fret + num);
    } else {

```

```

// 可能得到两个音高一样但标记方式不一样的音
return nextKey.map((key) => {
    return new this.constructor((octave 0 ? octaveString : '
') + key + (octave > 0 ? octaveString : ''), this.position.string, this.position.fret + num);
});
}
} else {
    return null;
}
}
}
}

```

有了这个单音类后，后续可以借用它来方便地对比两个音之间的跨度，并且可以通过构建吉他每根弦的初始音，通过 `step` 方法推导出吉他其他任意位置的音高。

执行示例：

创建一个 `1 (do)` 的单音实例

```

> let tone = new Tone("1")
< undefined
> console.log(tone)
▼ Tone {syllableMap: Array(7), keyMap: Array(12), intervalMap: Array(12), toneString: "1", toneNormal: "1", ...}
  flat: ""
  ▶ intervalMap: (12) ["C", Array(2), "D", Array(2), "E", "F", Array(2), "G", Array(2), "A", Array(2), "B"]
  key: "1"
  ▶ keyMap: (12) ["1", Array(2), "2", Array(2), "3", "4", Array(2), "5", Array(2), "6", Array(2), "7"]
  octave: 0
  ▶ position: {string: undefined, fret: undefined}
  sharp: ""
  ▶ syllableMap: (7) ["do", "re", "mi", "fa", "sol", "la", "si"]
  syllableName: "do"
  toneNormal: "1"
  toneString: "1"
  ▶ __proto__: Object
< undefined
> tone.findKeyIndex("#5")
< 8
> tone.findKeyIndex("2")
< 2
> tone.findKeyIndex("b7")
< 10
> tone.findKeyIndex("7")
< 11
> tone.findKeyIndex("1")
< 0

```

单音 `1 (do)`，往高跨 `5个半音`，得到单音 `4 (fa)`；往高跨 `6个半音`，得到两个音 `#4 (升fa)` 与 `b5 (降sol)`，这两个音处于同一音高，本质相同，只是标记方式不一样。

```
> tone.step(5)
◁ ▼ Tone {syllableMap: Array(7), keyMap: Array(12), intervalMap: Array(12), toneString: "4", toneNormal: "4", ...}
  flat: ""
  ▶ intervalMap: (12) ["C", Array(2), "D", Array(2), "E", "F", Array(2), "G", Array(2), "A", Array(2), "B"]
  key: "4"
  ▶ keyMap: (12) ["1", Array(2), "2", Array(2), "3", "4", Array(2), "5", Array(2), "6", Array(2), "7"]
  octave: 0
  ▶ position: {string: undefined, fret: NaN}
  sharp: ""
  ▶ syllableMap: (7) ["do", "re", "mi", "fa", "sol", "la", "si"]
  syllableName: "fa"
  toneNormal: "4"
  toneString: "4"
  ▶ __proto__: Object

> tone.step(6)
◁ ▼ (2) [Tone, Tone]
  ▼ 0: Tone
    flat: ""
    ▶ intervalMap: (12) ["C", Array(2), "D", Array(2), "E", "F", Array(2), "G", Array(2), "A", Array(2), "B"]
    key: "4"
    ▶ keyMap: (12) ["1", Array(2), "2", Array(2), "3", "4", Array(2), "5", Array(2), "6", Array(2), "7"]
    octave: 0
    ▶ position: {string: undefined, fret: NaN}
    sharp: "#4"
    ▶ syllableMap: (7) ["do", "re", "mi", "fa", "sol", "la", "si"]
    syllableName: "fa"
    toneNormal: "#4"
    toneString: "#4"
    ▶ __proto__: Object
  ▼ 1: Tone
    flat: "b"
    ▶ intervalMap: (12) ["C", Array(2), "D", Array(2), "E", "F", Array(2), "G", Array(2), "A", Array(2), "B"]
    key: "5"
    ▶ keyMap: (12) ["1", Array(2), "2", Array(2), "3", "4", Array(2), "5", Array(2), "6", Array(2), "7"]
    octave: 0
    ▶ position: {string: undefined, fret: NaN}
    sharp: ""
    ▶ syllableMap: (7) ["do", "re", "mi", "fa", "sol", "la", "si"]
    syllableName: "sol"
    toneNormal: "b5"
    toneString: "b5"
    ▶ __proto__: Object
    length: 2
  ▶ __proto__: Array(0)
```

## 二、和弦命名推导

## 1. 什么是和弦

先上个百度词条：

[hé xián] ★ 收藏 👍 2590 🔗 160

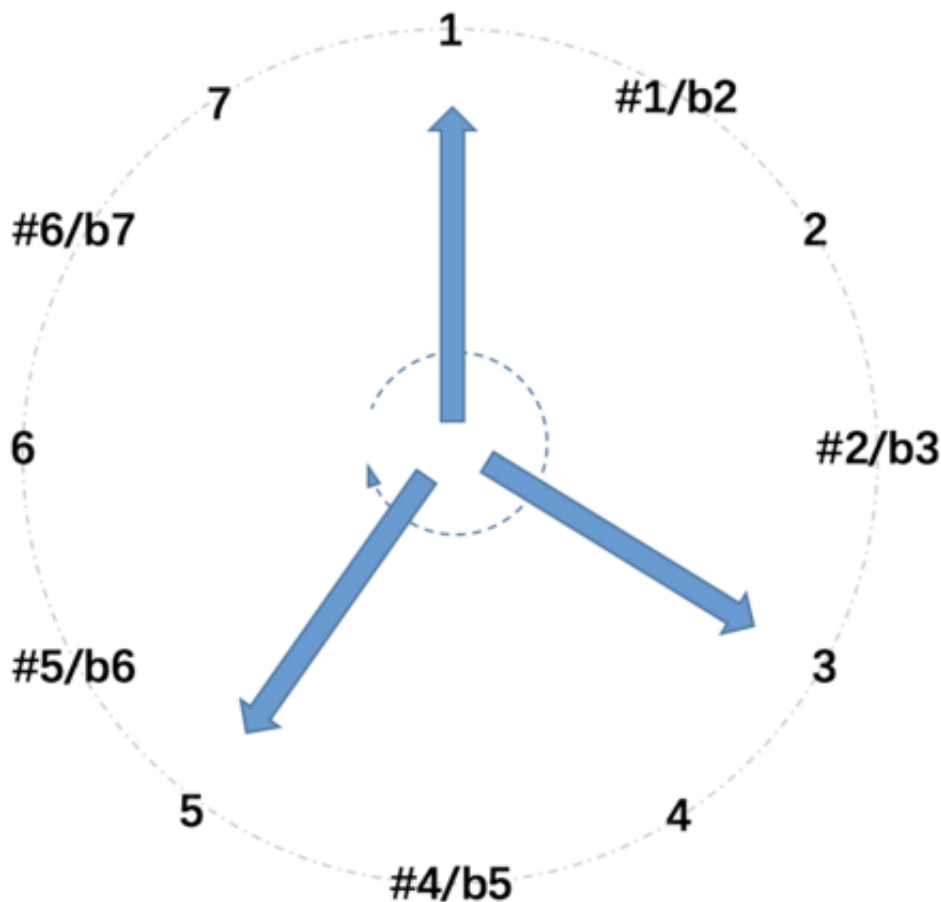
## 和弦 (乐理概念) ✎ 编辑

和弦（Chord）源自希臘文χορδή，是**乐理**上的一个概念，指的是一定**音程**关系的一组声音。将三个或以上的音，按照三度或非三度的叠置关系，在纵向上加以结合，就成为和弦。

由此白话提炼和弦的三个要素：

- (1) 由三个或三个以上的音构成；
- (2) 音之间有跨度关系（三度或非三度）；
- (3) 音之间要从低到高排列。

由此我画了一张图：



一个音程上的 12个音 可以像时钟的刻度那样排列， 顺时针 方向代表 音的从低到高 ； 然后我们将“时针”、“分针”、“秒针”在不重叠且相互有一定间隔的情况下随意拨弄，把他们指向的音顺时针连起来，就 可能 构成了一个三个音组成的 和弦

（同理更多音组成的和弦就相当于再往里加指针）。

这样一看，便能发现这更像一个 排列组合 问题，拿三个音的组合来说，从12个音里面任意挑3个音（不排序），会有 220 种情况，但这里面并不都是 和弦 ； 和弦和弦，顾名思义，听起来得和谐得不难听，这开始更像是人们的主观意识判断，但随着音乐知识体系的成熟，和弦也会有一套公认的标准，变得向数学公式那样有迹可循。

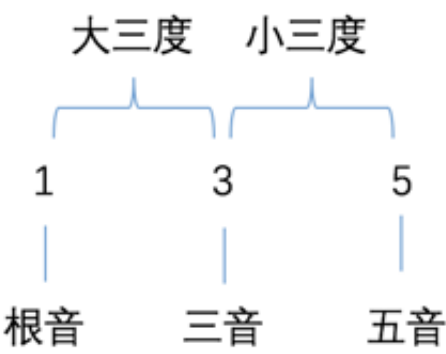
细想一下，一个和弦好不好听，带什么感情色彩，取决于组成音的相互映衬关系，也就是音之间的相互音高间隔，隔得太近会别扭，隔得太远也别扭，那就得取个适中的，这个适中就是 三度 ；

三度 又分为 大三度 与 小三度

大三度 ： 两个 全音 的跨度，即 4 个 半音 的跨度。

小三度 ： 一个 全音 加一个 半音 的跨度，即 3 个 半音 的跨度。

C调下的 C和弦 组成音如下：



对照上图那个刻度盘可数出来：

1 (do) 与 3 (mi) 中间还夹了 #1/b2 、 2 、 #2/b3 这3个音，共4个半音的跨度；

3 (mi) 与 5 (sol) 中间还夹了 4 、 #4/b5 这2个音，共3个半音的跨度；

那么像这样组成的和弦就成为 大三和弦 。

## 2. 常见和弦标记规则

和弦类型	组成	标记
大三和弦	大三度 + 小三度	
小三和弦	小三度 + 大三度	m
增三和弦	大三度 + 大三度	aug
减三和弦	小三度 + 小三度	dim
大小七和弦（属七和弦）	大三和弦 + 小三度	7 或 Mm7
大大七和弦（大七和弦）	大三和弦 + 大三度	maj7 或 M7
小小七和弦（小七和弦）	小三和弦 + 小三度	m7 或 mm7
小大七和弦	小三和弦 + 大三度	mM7
减七和弦	减三和弦 + 小三度	dim7
半减七和弦	减三和弦 + 大三度	m7-5
增属七和弦	增三和弦 + 减三度	7#5 或 M7+5
增大七和弦	增三和弦 + 小三度	aug7 或 Maj7#5



加音和弦 与 指定和弦根音 相对复杂些，暂不讨论。

### 3. 和弦根音

和弦组成音中的第一个音为和弦的 根音 ，也叫 基础音 ，可以根据当前的 调式 和某和弦的 根音 来判断该和弦的初始名称，例如在 C调 下， 根音 与 和弦名 的对照关系如下：

I	II	III	IV	V	VI	VII
1	2	3	4	5	6	7
C	D	E	F	G	A	B

通俗点说相当于，在 某调 下，一个和弦的根音为该调的 1 (do) 时，那它就叫 某和弦 （额外标记根据音之间的三度关系再添加），例如：

C调下：

根音为 1 (do) 构成的和弦名为 C ；

根音为 2 (re) 构成的和弦名为 D ；

D调下：

根音为 1 (do) 构成的和弦名为 D ；

根音为 1 (do) 构成的和弦名为 E ；

B调下：

根音为 1 (do) 构成的和弦名为 B ；

根音为 2 (do) 构成的和弦名为 C ；

### 4. 和弦完整名称计算

基于以上的乐理规则，可以实现如下推导和弦名的类：  
...

// 和弦名称推导

```
class ChordName {
  constructor(chordTone) {
    // 实例化一个单音类做工具，用来计算音与各种标记的映射关系
    this.toneUtil = new Tone();
  }
}
```

```
// 获取两个音的间隔跨度
getToneSpace(tonePre, toneNext) {
let toneSpace = this.toneUtil.findKeyIndex(toneNext) - this.toneUtil.findKeyIndex(tonePre);

return toneSpace = toneSpace < 0 ? toneSpace + 12 : toneSpace;
}

// 大三度
isMajorThird(tonePre, toneNext) {
return this.getToneSpace(tonePre, toneNext) === 4;
}

// 小三度
isMinorThird(tonePre, toneNext) {
return this.getToneSpace(tonePre, toneNext) === 3;
}

// 增三度
isMajorMajorThird(tonePre, toneNext) {
return this.getToneSpace(tonePre, toneNext) === 5;
}

// 减三度
isMinorMinorThird(tonePre, toneNext) {
return this.getToneSpace(tonePre, toneNext) === 2;
}

// 大三和弦
isMajorChord(chordTone) {
return this.isMajorThird(chordTone[0], chordTone[1]) && this.isMinorThird(chordTone[1],
chordTone[2]);
}

// 小三和弦 m
isMinorChord(chordTone) {
return this.isMinorThird(chordTone[0], chordTone[1]) && this.isMajorThird(chordTone[1],
chordTone[2]);
}

// 增三和弦 aug
isAugmentedChord(chordTone) {
return this.isMajorThird(chordTone[0], chordTone[1]) && this.isMajorThird(chordTone[1],
chordTone[2]);
}

// 减三和弦 dim
isDiminishedChord(chordTone) {
```

```
return this.isMinorThird(chordTone[0], chordTone[1]) && this.isMinorThird(chordTone[1],
chordTone[2]);
}
// 挂四和弦
isSus4(chordTone) {
return this.isMajorMajorThird(chordTone[0], chordTone[1]) &&
this.isMinorMinorThird(chordTone[1], chordTone[2]);
}
// 大小七和弦/属七和弦 7 / Mm7
isMajorMinorSeventhChord(chordTone) {
if (chordTone.length 4) return false;
return this.isMajorChord(chordTone) && this.isMinorThird(chordTone[2], chordTone[3]);
}
// 小大七和弦 mM7
isMinorMajorSeventhChord(chordTone) {
if (chordTone.length 4) return false;
return this.isMinorChord(chordTone) && this.isMajorThird(chordTone[2], chordTone[3]);
}
// 大七和弦 maj7 / M7
isMajorMajorSeventhChord(chordTone) {
if (chordTone.length 4) return false;
return this.isMajorChord(chordTone) && this.isMajorThird(chordTone[2], chordTone[3]);
}
// 小七和弦 m7 / mm7
isMinorMinorSeventhChord(chordTone) {
if (chordTone.length 4) return false;
return this.isMinorChord(chordTone) && this.isMinorThird(chordTone[2], chordTone[3]);
}
// 减七和弦 dim7
isDiminishedSeventhChord(chordTone) {
if (chordTone.length 4) return false;
return this.isDiminishedChord(chordTone) && this.isMinorThird(chordTone[2], chordTone[3]);
}
// 半减七和弦 m7-5
isHalfDiminishedSeventhChord(chordTone) {
if (chordTone.length 4) return false;
return this.isDiminishedChord(chordTone) && this.isMajorThird(chordTone[2], chordTone[3]);
}
// 增属七和弦 7#5 / M7+5
```

```

isHalfAugmentedSeventhChord(chordTone) {
  if (chordTone.length 4) return false;
  return this.isAugmentedChord(chordTone) && this.isMinorMinorThird(chordTone[2],
  chordTone[3]);
}
// 增大七和弦 aug7 / Maj7#5
isAugmentedSeventhChord(chordTone) {
  if (chordTone.length 4) return false;
  return this.isAugmentedChord(chordTone) && this.isMinorThird(chordTone[2], chordTone[3]);
}
// 获取音对应的根音和弦名
getKeyName(key) {
  let keyName = this.toneUtil.intervalMap[this.toneUtil.findKeyIndex(key)];
  if (is(keyName)('Array')) {
    keyName = /b/.test(key) ? keyName[1] : keyName[0];
  };
  return keyName;
}
// 计算和弦名
getChordName(chordTone) {
  let rootKey = chordTone[0];
  // 和弦的字母名
  let chordRootName = this.getKeyName(rootKey);
  // 和弦字母后面的具体修饰名
  let suffix = '...';
  let suffixArr = [];
  // 三音和弦的遍历方法及对应修饰名
  let chord3SuffixMap = [{
    fn: this.isMajorChord,
    suffix: ''
  }, {
    fn: this.isMinorChord,
    suffix: 'm'
  }, {
    fn: this.isAugmentedChord,
    suffix: 'aug'
  }, {
    fn: this.isDiminishedChord,
    suffix: 'dim'
  }];

```

```

}, {
  fn: this.isSus4,
  suffix: 'sus4'
}];
// 四音和弦的遍历方法及对应修饰名
let chord4SuffixMap = [{
  fn: this.isMajorMinorSeventhChord,
  suffix: '7'
}, {
  fn: this.isMinorMajorSeventhChord,
  suffix: 'mM7'
}, {
  fn: this.isMajorMajorSeventhChord,
  suffix: 'maj7'
}, {
  fn: this.isMinorMinorSeventhChord,
  suffix: 'm7'
}, {
  fn: this.isDiminishedSeventhChord,
  suffix: 'dim7'
}, {
  fn: this.isHalfDiminishedSeventhChord,
  suffix: 'm7-5'
}, {
  fn: this.isHalfAugmentedSeventhChord,
  suffix: '7#5'
}, {
  fn: this.isAugmentedSeventhChord,
  suffix: 'aug7'
}];
// 三音和弦
if (chordTone.length === 3) {
  suffixArr = chord3SuffixMap.filter((item) => {
    return item.fn.bind(this, chordTone)();
  });
  suffix = suffixArr.length > 0 ? suffixArr[0].suffix : suffix;
} else {
  // 四音和弦
  suffixArr = chord4SuffixMap.filter((item) => {

```

```
return item.fn.bind(this, chordTone());
});
suffix = suffixArr.length > 0 ? suffixArr[0].suffix : suffix;
}
// 拼接起来得到完整的和弦名
return chordRootName + suffix;
}
}
```

运行示例：



### ### 三、和弦指法推导

#### #### 1\．指法图

一个完整的吉他和弦指法图的例子如下，右边对照为真实的吉他：



说明下几个名词的意思：

`品位`：真实的吉他琴颈上被划分成了很多格子，当手指按在不同的格子上时，改变了对应琴弦振动的弦长，那么它的发音高低也会跟着改变，这些按住可以改变音高的格子就被称作`品位`，或`品格`（不得不说这“品”字取得真好）；

`品位标记`：在指法图的左侧标记了一个数字，表示图上该行的`品位`实际位于吉他上的`品位`是多少（所谓的相对坐标系与绝对坐标系）；

`空弦音`：即`第0品`，左手不用按标记对应的弦，右手直接拨它；

`和弦外音`：当某根弦的空弦音以及在它在该指法图范围内能产生的音都不属于该和弦的组成音时，那么这根弦应该禁止弹奏，故在该弦上面标记一个`“叉号”`；

`按弦手指标记`：用黑色的圆点标记手指按在各弦上的位置，最完整的指法图还会在上面加上数字`1`、`2`、`3`、`4`，分别代表`食指`、`中指`、`无名指`、`小拇指`，相当于哪根手指该放哪儿都告诉清楚了。

#### #### 2\．吉他弦上音的分布

我从网上抠来了这张带着历史气息的彩图：



可以观察到，同样一个音，在吉他弦上的位置可以有多个；而简单的和弦的组成音也就三四个，所以要想一下子从这些纵横的格子里寻出某个和弦所有可能的指法，同时还要考虑实际指法的各种约束：

比如你左手能用上的只有不超过5根手指头而弦有6根，但食指是可以使用大横按按多根弦的，但大横按只能按在该指法的最低品位上；还得考虑指法按弦后是包括了和弦里所有的音，同时相邻两弦的音不能一样...

诸如此类，想要一下子心算出来所有可能的结果，怕是为难我胖虎了。

不过这个很适合用递归算法解决。

#### #### 3\。指法推导

为此专门构建一个类，在初始化的时候使用之前写的单音类，算出吉他弦上所有位置的音。之后就可以通过 `this.toneMap[string][fret]` 的形式直接获得该位置的音，例如 `this.toneMap[1][3]` 获取 `1弦3品` 的音。

// 吉他和弦指法推导类

```
class GuitarChord {
```

```
  constructor() {
```

```
    // 暂定的吉他的最大品格数
```

```
    this.fretLength = 15;
```

```
    // 构建1到6弦的初始音
```

```
    this.initialTone = [
```

```
      new Tone('3.', 1, 0),
```

```
      new Tone('7', 2, 0),
```

```
      new Tone('5', 3, 0),
```

```
      new Tone('2', 4, 0),
```

```
      new Tone('.6', 5, 0),
```

```
      new Tone('.3', 6, 0)
```

```
    ];
```

```
    // 用于吉他上所有位置对应的音
```

```
    this.toneMap = [];
```

```
    // 从1到6弦，从品位数的低到高，依次计算每个位置的音
```

```
    for (let string = 1; string this.initialTone.length; string++) {
```

```
      this.toneMap[string] = [];
```

```
      for (let fret = 0; fret this.fretLength; fret++) {
```

```
        this.toneMap[string].push(this.initialTone[string - 1].step(fret));
```

```
      }
```

```
    }
```

```
  }
```

```
}
```

给它加上一个公用的单音位置搜寻方法：

```
// 在指定的品格数范围内，查找某个音在某根弦的音域下所有的品格位置
/*
 * @param key 搜寻的音（字符串形式）
 * @param toneArray 音域数组，即某根弦上所有单音类按顺序组成的数组
 * @param fretStart 搜寻的最低品格数
 * @param fretEnd 搜寻的最高品格数
 */
findFret(key, toneArray, fretStart, fretEnd) {
  key = key.replace(/./g, '');
  let fretArray = [];
  fretStart = fretStart ? fretStart : 0;
  fretEnd = fretEnd ? (fretEnd + 1) : toneArray.length;
  for (let i = fretStart; i if (is(toneArray[i])('Array')) {
    let toneStringArray = toneArray[i].map((item) => {
      return item.toneNormal;
    });
    if (toneStringArray.includes(key)) {
      fretArray.push(i);
    }
  } else {
    if (toneArray[i].toneString.replace(/./g, '') === key) {
      fretArray.push(i);
    }
  }
}
return fretArray;
}
...
```

接下来是核心的循环递归算法，先构思下大致的递归的流程：

(1) 指定从1弦开始，启动递归。（递归入口）

(2) 指定了某弦后，循环遍历和弦的组成音，计算是否有音落在该弦指定的品位范围内，如果没有，返回 `false`；如果有，转步骤（3）。

(3) 先 `保存` 该音与它的按弦位置，当前位置最终有效取决于，`当且仅当` 在它后面的所有弦



也是能找到按弦位置的有效解，如果该弦是第6弦，返回 `true` ，递归结束（递归出口），否则转步骤（4）；

（4） 当前结果最终的有效性 = 当前临时结果有效性（`true`） && 下一根弦是否存在有效解（此时已转至步骤（3））。若当前结果最终有效，返回 `true`；若无效，回退`pop` 出之前在该弦保存的结果。

最后实现还需考虑相邻两弦音不能相同，另外为了便于回溯整体结果，在单次的结果保存时，添加了指向上一次结果的指针 `pre` 。

```
// 递归遍历范围内的指定和弦的所有位置组合
/*
 * @param stringIndex 当前遍历到的弦的序号
 * @param toneIndex 上一根弦使用的音的序号（用于相邻的两根弦的音不重复）
 * @param fretStart 遍历的最低品格数
 * @param fretEnd 遍历的最高品格数
 * @param preResult 上一根弦确定的音的结果
 * @param positionSave 保存该轮递归的结果
 */
calc(stringIndex, toneIndex, fretStart, fretEnd, preResult, positionSave) {
  let toneArray = this.toneMap[stringIndex];
  let result = false;
  // 从和弦音的数组里逐个选出音进行试探（this.chordTone在后面提到的函数中赋值）
  for (let i = 0; i < this.chordTone.length; i++) {
    // 相邻的上一根弦已使用的音不做本次计算
    if (i !== toneIndex) {
      let resultNext = false;
      let toneKey = this.chordTone[i];
      // 在品格范围内查找当前音的位置
      let fret = this.findFret(toneKey, toneArray, fretStart, fretEnd);
      // 品格范围内存在该音
      if (fret.length > 0) {
        // 记录该音的位置，几弦几品与音的数字描述
        let resultNow = {
          string: stringIndex,
          fret: fret[0],
          key: toneKey
        }
        // 在本次记录上保存上一根弦的结果，方便回溯
      }
    }
  }
}
```

```

resultNow.pre = preResult ? preResult : null;
// 保存本次结果
positionSave.push(resultNow);
// 设置该弦上的结果标记
resultNext = true;
// 没有遍历完所有6根弦，则继续往下一根弦计算，附带本次的结果记录
if (stringIndex < this.initialTone.length) {
    let nextStringIndex = stringIndex + 1;
    // 该弦上的结果的有效标记，取决于它后面的弦的结果均有效
    resultNext = resultNext && this.calc(nextStringIndex, i, fretStart, fretEnd,
    resultNow, positionSave);
} else {
    // 所有弦均遍历成功，代表递归结果有效
    resultNext = true;
}
// 在该弦的计算结果无效，吐出之前保存的该弦结果
if (!resultNext) {
    positionSave.pop();
}
} else {
    // 品格范围内不存在该音
    resultNext = false;
}
// 任意一个和弦里的音，能在该弦取得有效结果，则该弦上的结果有效
result = result || resultNext;
};
return result;
}

```

使用此递归方法，用 1 、 3 、 5 为和弦组成音做输入，会得到类似下面这样的结果：

```
▶ 0: {string: 1, fret: 0, key: "3", pre: null}
▶ 1: {string: 2, fret: 1, key: "1", pre: {...}}
▶ 2: {string: 3, fret: 0, key: "5", pre: {...}}
▶ 3: {string: 4, fret: 2, key: "3", pre: {...}}
▶ 4: {string: 5, fret: 3, key: "1", pre: {...}}
▶ 5: {string: 6, fret: 0, key: "3", pre: {...}}
▶ 6: {string: 6, fret: 3, key: "5", pre: {...}}
▶ 7: {string: 1, fret: 3, key: "5", pre: null}
▶ 8: {string: 2, fret: 1, key: "1", pre: {...}}
▶ 9: {string: 3, fret: 0, key: "5", pre: {...}}
▶ 10: {string: 4, fret: 2, key: "3", pre: {...}}
▶ 11: {string: 5, fret: 3, key: "1", pre: {...}}
▶ 12: {string: 6, fret: 0, key: "3", pre: {...}}
▶ 13: {string: 6, fret: 3, key: "5", pre: {...}}
▶ 14: {string: 2, fret: 5, key: "3", pre: {...}}
▶ 15: {string: 3, fret: 5, key: "1", pre: {...}}
▶ 16: {string: 4, fret: 2, key: "3", pre: {...}}
▶ 17: {string: 5, fret: 3, key: "1", pre: {...}}
▶ 18: {string: 6, fret: 0, key: "3", pre: {...}}
▶ 19: {string: 6, fret: 3, key: "5", pre: {...}}
▶ 20: {string: 4, fret: 5, key: "5", pre: {...}}
▶ 21: {string: 5, fret: 3, key: "1", pre: {...}}
▶ 22: {string: 6, fret: 0, key: "3", pre: {...}}
▶ 23: {string: 6, fret: 3, key: "5", pre: {...}}
▶ 24: {string: 3, fret: 0, key: "5", pre: {...}}
▶ 25: {string: 4, fret: 2, key: "3", pre: {...}}
▶ 26: {string: 5, fret: 3, key: "1", pre: {...}}
▶ 27: {string: 6, fret: 0, key: "3", pre: {...}}
▶ 28: {string: 6, fret: 3, key: "5", pre: {...}}
```

递归在执行的时候，在每个节点上可能产生多个分支节点层层往下深入，以上的打印其实就是列出了每个节点的数据。而我们需要的是将这个递归结果拆分为不同指法结果的数组，就像下面这样：

```

▼ 0: Array(6)
  ▶ 0: {string: 1, fret: 0, key: "3"}
  ▶ 1: {string: 2, fret: 1, key: "1"}
  ▶ 2: {string: 3, fret: 0, key: "5"}
  ▶ 3: {string: 4, fret: 2, key: "3"}
  ▶ 4: {string: 5, fret: 3, key: "1"}
  ▶ 5: {string: 6, fret: 0, key: "3"}
  length: 6
  ▶ __proto__: Array(0)
▼ 1: Array(6)
  ▶ 0: {string: 1, fret: 0, key: "3"}
  ▶ 1: {string: 2, fret: 1, key: "1"}
  ▶ 2: {string: 3, fret: 0, key: "5"}
  ▶ 3: {string: 4, fret: 2, key: "3"}
  ▶ 4: {string: 5, fret: 3, key: "1"}
  ▶ 5: {string: 6, fret: 3, key: "5"}
  length: 6
  ▶ __proto__: Array(0)
▼ 2: Array(6)
  ▶ 0: {string: 1, fret: 3, key: "5"}
  ▶ 1: {string: 2, fret: 1, key: "1"}
  ▶ 2: {string: 3, fret: 0, key: "5"}
  ▶ 3: {string: 4, fret: 2, key: "3"}
  ▶ 4: {string: 5, fret: 3, key: "1"}
  ▶ 5: {string: 6, fret: 0, key: "3"}
  length: 6
  ▶ __proto__: Array(0)
▶ 3: (6) [{...}, {...}, {...}, {...}, {...}, {...}]
▶ 4: (6) [{...}, {...}, {...}, {...}, {...}, {...}]
▶ 5: (6) [{...}, {...}, {...}, {...}, {...}, {...}]
▶ 6: (6) [{...}, {...}, {...}, {...}, {...}, {...}]
▶ 7: (6) [{...}, {...}, {...}, {...}, {...}, {...}]
▶ 8: (6) [{...}, {...}, {...}, {...}, {...}, {...}]
▶ 9: (6) [{...}, {...}, {...}, {...}, {...}, {...}]

```

为此添加一个 `filter`

函数：

```

// 和弦指法过滤器
filter(positionSave) {
// 从6弦开始回溯记录的和弦指法结果，拆解出所有指法组合
let allResult = positionSave.filter((item) => {
return item.string === this.initialTone.length
}).map((item) => {

```

```

let resultItem = [{
string: item.string,
fret: item.fret,
key: item.key
}];
while (item.pre) {
item = item.pre;
resultItem.unshift({
string: item.string,
fret: item.fret,
key: item.key
});
}
return resultItem;
});
if (allResult.length > 0) {
// 依次调用各个过滤器
return this.integrityFilter(this.fingerFilter(this.rootToneFilter(allResult)));
} else {
return [];
}
}
}

```

可以看到回溯计算出理想的结果形式后，末尾还调用了 `多个过滤器`，因为代码计算出的符合组成音的所有指法组合，可能并不符合真实的按弦情况，需要进行多重的过滤。

## 4. 指法过滤

- `根音条件过滤`

例如以 `1`、`3`、`5` 作为和弦音，根音为 `1`，而初步得到的结果可能如下：

```

▼ 0: Array(6)
  ► 0: {string: 1, fret: 0, key: "3"}
  ► 1: {string: 2, fret: 1, key: "1"}
  ► 2: {string: 3, fret: 0, key: "5"}
  ► 3: {string: 4, fret: 2, key: "3"}
  ► 4: {string: 5, fret: 3, key: "1"}
  ► 5: {string: 6, fret: 0, key: "3"}
  length: 6
  ► __proto__: Array(0)
▼ 1: Array(6)
  ► 0: {string: 1, fret: 0, key: "3"}
  ► 1: {string: 2, fret: 1, key: "1"}
  ► 2: {string: 3, fret: 0, key: "5"}
  ► 3: {string: 4, fret: 2, key: "3"}
  ► 4: {string: 5, fret: 3, key: "1"}
  ► 5: {string: 6, fret: 3, key: "5"}
  length: 6
  ► __proto__: Array(0)

```

而一个和弦在吉他上弹奏时，根音应该为所有发声的音中最低的音，上图中最低的音要么是 6弦0品 的 3，要么是位于 6弦3品 的 5，不符合要求，而 5弦3品 刚好是该和弦根音，故应该 禁用第6弦 （这里的禁用是将该弦的按弦品位 fret 标记为 null

)

```

// 根音条件过滤
rootToneFilter(preResult) {
  let nextResult = new Set();
  preResult.forEach((item) => {
    // 允许发声的弦的总数，初始为6
    let realStringLength = 6;
    // 从低音弦到高音弦遍历，不符合根音条件则禁止其发声
    for (var i = item.length - 1; i >= 0; i--) {
      if (item[i].key !== this.rootTone) {
        item[i].fret = null;
        item[i].key = null;
        realStringLength--;
      } else {
        break;
      }
    }
  });
  return nextResult;
}

```

```

    }
  }
  if (realStringLength >= 4) {
    // 去重复
    nextResult.add(JSON.stringify(item));
  }
});
// 去重后的Set解析成对应数组返回
return [...nextResult].map(item => JSON.parse(item));
}

```

- 按弦手指数量过滤

左手按弦的时候，一般最多只能用上4个手指（大拇指极少用到），而用递归方法算出的结果，可能包含了各种奇奇怪怪的按法，比如下面这个：

```

▼ 5: Array(6)
  ► 0: {string: 1, fret: 3, key: "5"}
  ► 1: {string: 2, fret: 5, key: "3"}
  ► 2: {string: 3, fret: 5, key: "1"}
  ► 3: {string: 4, fret: 2, key: "3"}
  ► 4: {string: 5, fret: 3, key: "1"}
  ► 5: {string: 6, fret: 3, key: "5"}
  length: 6
  ► __proto__: Array(0)

```

看上去包含了和弦的所有组成音，但是就算经过上一轮的过滤禁用了第6弦，每个 非0 的品位都需要用手指去按，这样算下来也需要 5

个手指，故类似这样的结果都应该二次过滤掉：

```

// 按弦手指数量过滤
fingerFilter(preResult) {
  return preResult.filter((chordItem) => {
    // 按弦的最小品位
    let minFret = Math.min.apply(null, chordItem.map(item => item.fret).filter(fret => (fret !== null)));
    // 记录需要的手指数
    let fingerNum = minFret > 0 ? 1 : 0;
    chordItem.forEach((item) => {
      if (item.fret !== null && item.fret > minFret) {
        fingerNum++;
      }
    });
  });
}

```

```

        return fingerNum 4;
    });
}

```

- 和弦组成音完整性过滤

递归计算所有可能的指法组合时，虽然保证了相邻两个音不重复，但不保证所有的和弦组成音都被使用了，而且在新一轮根音过滤时，可能禁用了部分弦的发声，这可能导致丢掉了其中唯一一个组成音，所以最后还需进行一轮完整性过滤，剔除残次品：

```

// 和弦组成音完整性过滤
integrityFilter(preResult) {
    return preResult.filter((chordItem) => {
        let keyCount = [...new Set(chordItem.map(item => item.key).filter(key => key != null))].length;
        return keyCount === this.chordTone.length;
    });
}

```

## 5. 指法计算入口

由这里输入和弦的组成音，计算这些音所有可能出现的品格位置，然后从低到高，依次计算4或5个品格范围内的和弦指法，经整合过滤后得到该和弦所有的位置的正确指法。

```

// 和弦指法计算入口
chord() {
    let chordTone;
    if (is(arguments[0])('Array')) {
        chordTone = arguments[0];
    } else {
        chordTone = Array.prototype.slice.apply(arguments).map((item) => {
            let tone = new Tone(item.toString());
            return tone.flat + tone.sharp + tone.key;
        });
    }
    // 和弦组成音
    this.chordTone = chordTone;
    // 根音
    this.rootTone = chordTone[0];
    this.chordResult = [];
    let fretArray = [];
    // 查找和弦里的音可能存在的品格位置，保存至fretArray
    chordTone.forEach((item) => {
        for (let i = 1; i <= this.toneMap.length; i++) {

```



```

        fretArray = fretArray.concat(this.findFret(item, this.toneMap[i]
));
    }
});
fretArray = [...new Set(fretArray)];
// 品格位置从小到大排序
fretArray.sort((a, b) => {
    return a - b;
});
// 从低把位到高把位，计算范围内的所有该和弦指法
for (let i = 0; i < fretArray.length; i++) {
    let fretStart = fretArray[i];
    // 在不需要使用大横按时，即在最低的把位计算时，可把计算的品格范围扩大一格
    let fretEnd = fretStart > 0 ? (fretStart + 4) : (fretStart + 5);
    // 最高范围不能超过吉他的最高品格数
    if (fretEnd < this.fretLength) {
        let positionSave = [];
        // 从1弦开始启动递归计算
        if (this.calc(1, null, fretStart, fretEnd, null, positionSave))
        {
            // 单次结果过滤并保存
            this.chordResult.push(...this.filter(positionSave));
        }
    }
}
// 结果去重
let result = [...new Set(this.chordResult.map(item => JSON.stringify(item)))]
    .map(item => JSON.parse(item));
return result;
}

```

运行示例：

```

> let chord = new GuitarChord();
let chordTone = ['1', '3', '5'];
chord.chord(chordTone);
< ▼ (9) [Array(6), Array(6), Array(6), Array(6), Array(6), Array(6), Array(6), Array(6)] ⓘ
  ▼ 0: Array(6)
    ▶ 0: {string: 1, fret: 0, key: "3"}
    ▶ 1: {string: 2, fret: 1, key: "1"}
    ▶ 2: {string: 3, fret: 0, key: "5"}
    ▶ 3: {string: 4, fret: 2, key: "3"}
    ▶ 4: {string: 5, fret: 3, key: "1"}
    ▶ 5: {string: 6, fret: null, key: null}
    length: 6
    ▶ __proto__: Array(0)
  ▶ 1: (6) [_, _, _, _, _, _]
  ▼ 2: Array(6)
    ▶ 0: {string: 1, fret: 3, key: "5"}
    ▶ 1: {string: 2, fret: 5, key: "3"}
    ▶ 2: {string: 3, fret: 5, key: "1"}
    ▶ 3: {string: 4, fret: 5, key: "5"}
    ▶ 4: {string: 5, fret: 3, key: "1"}
    ▶ 5: {string: 6, fret: null, key: null}
    length: 6
    ▶ __proto__: Array(0)
  ▶ 3: (6) [_, _, _, _, _, _]
  ▼ 4: Array(6)
    ▶ 0: {string: 1, fret: 8, key: "1"}
    ▶ 1: {string: 2, fret: 5, key: "3"}
    ▶ 2: {string: 3, fret: 5, key: "1"}
    ▶ 3: {string: 4, fret: 5, key: "5"}
    ▶ 4: {string: 5, fret: 7, key: "3"}
    ▶ 5: {string: 6, fret: 8, key: "1"}
    length: 6
    ▶ __proto__: Array(0)
  ▼ 5: Array(6)
    ▶ 0: {string: 1, fret: 8, key: "1"}
    ▶ 1: {string: 2, fret: 8, key: "5"}
    ▶ 2: {string: 3, fret: 9, key: "3"}
    ▶ 3: {string: 4, fret: 10, key: "1"}
    ▶ 4: {string: 5, fret: 10, key: "5"}
    ▶ 5: {string: 6, fret: 8, key: "1"}
    length: 6
    ▶ __proto__: Array(0)
  ▶ 6: (6) [_, _, _, _, _, _]
  ▶ 7: (6) [_, _, _, _, _, _]
  ▶ 8: (6) [_, _, _, _, _, _]
    length: 9
    ▶ __proto__: Array(0)

```

### 三、和弦指法结果可视化

特意挑选了svg作图，因为之前不会，借此机会学习了一下。

一个较为完整的和弦指法图，svg的代码示例如下（把这个扔到自己的html里打开也能直观看到结果）：

```

svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/x
link" version="1.1" id="svg" width="200" height="220" viewBox="0 0 150 150"
preserveAspectRatio="xMidYMin meet">
  defs>
    g id="forbidden">
      path d="M-5 -5 L5 5 M-5 5 L5 -5" stroke="#666" stroke-width="1"
fill="none"/>
    g>
    g id="blank_circle">
      circle cx="0" cy="0" r="6" stroke="#666" stroke-width="1" fill="

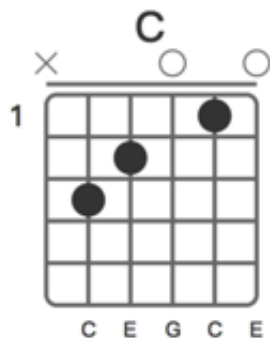
```

```

none"/>
  g>
  g id="block_circle">
    circle cx="0" cy="0" r="8" fill="#333"/>
  g>
  defs>
    rect x="25" y="45" rx="5" ry="5" width="100" height="100" style="fill:none
;stroke:#666;stroke-width:2"/>
    path d="M25 65 L125 65 M25 85 L125 85 M25 105 L125 105 M25 125 L125 125
M45 45 L45 145 M65 45 L65 145 M85 45 L85 145 M105 45 L105 145 M25 40 L125 40
" stroke="#666" stroke-width="2" fill="none"/>
    use xlink:href="#forbidden" x="25" y="30" />
    use xlink:href="#blank_circle" x="125" y="30" />
    use xlink:href="#blank_circle" x="85" y="30" />
    use xlink:href="#block_circle" x="105" y="55" />
    use xlink:href="#block_circle" x="65" y="75" />
    use xlink:href="#block_circle" x="45" y="95" />
    text x="67" y="20" fill="#333" font-size="20" font-weight="700">Ctext>
    text x="41.5" y="160" fill="#333" font-size="10" font-weight="700">Ctext
>
    text x="61.5" y="160" fill="#333" font-size="10" font-weight="700">Etext
>
    text x="81.5" y="160" fill="#333" font-size="10" font-weight="700">Gtext
>
    text x="101.5" y="160" fill="#333" font-size="10" font-weight="700">Ctex
t>
    text x="121.5" y="160" fill="#333" font-size="10" font-weight="700">Etex
t>
    text x="8" y="60" font-size="14" font-weight="700" fill="#333">1text>
svg>

```

显示效果如下：



当然了，得设计出一套可以画任意svg指法图的方案。

简单来说，就是将指法图拆分为多个子元素，有的画 网格 ，有的画 按弦位置 ，有的画 空弦符号 ，诸如此类，然后根据传入的指法结果，动态创建这些子元素加入svg即可；但需特别考虑各个元素可能会动态改变的位置，以及对于 大横按 的绘图处理。

这里代码我一擦全摆出来了，带了较为详尽的注释，就不细讲了（打字打累了...）

```
// 和弦svg绘图
```

```
class ChordSvg {
  constructor() {
    this.SVG_NS = "http://www.w3.org/2000/svg";
    this.XLINK_NS = "http://www.w3.org/1999/xlink";
    this.ATTR_MAP = {
      "className": "class",
      "svgHref": "href"
    };
    this.NS_MAP = {
      "svgHref": this.XLINK_NS
    };
    this.initChordSvg();
    this.minFret = 0;
  }

  // 创建svg相关元素
  createSVG(tag, attributes) {
    let elem = document.createElementNS(this.SVG_NS, tag);
    for (let attribute in attributes) {
      let name = (attribute in this.ATTR_MAP ? this.ATTR_MAP[attribute] : attribute);
      let value = attributes[attribute];
      if (attribute in this.NS_MAP) {
        elem.setAttributeNS(this.NS_MAP[attribute], name, value);
      } else {
        elem.setAttribute(name, value);
      }
    }
    return elem;
  }

  // 创建use标签
  createUse(href, x, y) {
    return this.createSVG('use', {
      svgHref: href,
      x: x,
      y: y
    });
  }

  // 设置禁止弹奏的叉号位置，位于几弦
  setForbidden(svg, string = 6) {
    svg.appendChild(this.createUse('#forbidden', 25 + 20 * (6 - string),
```

```

30));
}
// 设置空弦弹奏的空心圈位置, 位于几弦
setOpen(svg, string = 6) {
    svg.appendChild(this.createUse('#blank_circle', 25 + 20 * (6 - string), 30));
}
// 设置指法按弦位置, 几弦几品
setFinger(svg, string = 6, fret = 0) {
    if (+fret > 0 && +fret 5) {
        svg.appendChild(this.createUse('#block_circle', 25 + 20 * (6 - string), 35 + 20 * fret));
    }
}
// 设置大横按位置
setBarre(svg, stringTo, fret, barreFret) {
    if (fret > 0 && fret 5) {
        svg.appendChild(this.createSVG('rect', {
            className: 'chord-barre',
            width: stringTo * 20,
            x: 15 + 20 * (6 - stringTo),
            y: 27 + 20 * fret,
            rx: 8,
            ry: 8
        }));
    }
}
// 设置把位偏移的数字提示
setFretOffset(svg, fret, fretOffset, isBarreCover) {
    if (fret > 0) {
        let text = this.createSVG('text', {
            className: 'chord-barre-fret',
            x: isBarreCover ? 1 : 8,
            y: 40 + fret * 20
        });
        text.innerHTML = fretOffset;
        svg.appendChild(text);
    }
}
// 设置每根弦在按住和弦后的发音名
setStringKey(svg, string, keyName) {
    let xFixed = keyName.length === 2 ? -4 : 0;
    let text = this.createSVG('text', {
        className: 'chord-string-key',
        x: 21.5 + 20 * (6 - string) + xFixed,
        y: 160
    });
    text.innerHTML = keyName;
}

```

```

        svg.appendChild(text);
    }
    // 设置和弦名称
    setChordName(svg, name = '') {
        let xFixed = /\.\.\.\./ .test(name) ? 10 : 0;
        let text = this.createSVG('text', {
            className: 'chord-name',
            x: 75 - name.toString().length * 7 + xFixed,
            y: 20
        });
        text.innerHTML = name;
        svg.appendChild(text);
    }
    // 初始化svg
    initChordSvg() {
        // svg元素
        this.svg = this.createSVG('svg', {
            className: 'chord-svg',
            viewBox: '0 0 150 150',
            preserveAspectRatio: 'xMidYMin meet'
        });
        // 和弦图方块
        this.chordRect = this.createSVG('rect', {
            className: 'chord-rect',
            x: 25,
            y: 45,
            rx: 5,
            ry: 5
        });
        // 和弦网格, 代表弦和品
        this.chordGrid = this.createSVG('path', {
            className: 'chord-grid',
            d: 'M25 65 L125 65 M25 85 L125 85 M25 105 L125 105 M25 125 L125
125 M45 45 L45 145 M65 45 L65 145 M85 45 L85 145 M105 45 L105 145 M25 40 L125 40'
        });
        // 用于放置可复用的svg元素
        this.defs = this.createSVG('defs');
        // 禁止按弦的叉号标志
        this.g_forbidden = this.createSVG('g', {
            id: 'forbidden'
        });
        this.g_forbidden.appendChild(this.createSVG('path', {
            className: 'chord-forbidden',
            d: 'M-5 -5 L5 5 M5 5 L5 -5'
        }));
        // 空弦弹奏的空心圈标志
        this.g_blank_circle = this.createSVG('g', {

```

```

        id: 'blank_circle',
    });
    this.g_blank_circle.appendChild(this.createSVG('circle', {
        className: 'chord-blank-circle',
        cx: 0,
        cy: 0,
        r: 6
    }));
    // 表示按弦位置的实心圈标志
    this.g_block_circle = this.createSVG('g', {
        id: 'block_circle'
    });
    this.g_block_circle.appendChild(this.createSVG('circle', {
        className: 'chord-block-circle',
        cx: 0,
        cy: 0,
        r: 8
    }));
    // 可复用元素加入
    this.defs.appendChild(this.g_forbidden);
    this.defs.appendChild(this.g_blank_circle);
    this.defs.appendChild(this.g_block_circle);
    // svg子元素加入
    this.svg.appendChild(this.chordRect);
    this.svg.appendChild(this.chordGird);
    this.svg.appendChild(this.defs);
}
// 绘制和弦svg图案
/*
 * @param chordTone 和弦组成音数组
 * @param chord 和弦指法结果
 * @param target svg指法图dom容器
 */
drawChord(chordTone, chord, target) {
    let svg = this.svg.cloneNode(true);
    let fretArr = chord.map(item => item.fret).filter(fret => (fret != null));
    // 和弦指法中出现的最高品格位置
    let maxFret = Math.max.apply(null, fretArr);
    // 和弦指法中出现的最低品位位置
    let minFret = Math.min.apply(null, fretArr);
    // svg指法图案的起始品格位置相对于吉他上0品位置的偏移量
    let fretOffset = maxFret > 5 ? 0 : minFret;
    // 记录指法最低品位可能需要大横按的按弦数
    let barreCount = 0;
    // 大横按初始只横跨1弦到1弦（相当于没横按）
    let barreStringTo = 1;
    // 实例化用于计算和弦名称的类

```

```

let chordName = new ChordName();
// 遍历和弦指法数组
chord.forEach((item) => {
    if (item.fret == null) {
        // 某根弦没标记品格位置时禁止该弦弹奏
        this.setForbidden(svg, item.string);
    } else if (item.fret === 0) {
        // 某根弦没标记的品格位置为0品时标记空弦弹奏
        this.setOpen(svg, item.string);
    } else {
        // 剩下的指法绘制其对应的按法位置
        this.setFinger(svg, item.string, fretOffset > 0 ? item.fret
- fretOffset + 1 : item.fret);
    }
    // 当按在该和弦的最低品格位置的指法反复出现时
    if (item.fret === minFret) {
        // 计算大横按的跨度
        barreStringTo = item.string > barreStringTo ? item.string :
barreStringTo;
        // 计算大横按实际按弦的数量
        barreCount++;
    }
    // 在允许弹奏的弦的下方标记其对应的音名
    if (item.fret != null) {
        this.setStringKey(svg, item.string, chordName.getKeyName(item.key));
    }
});
// 将真实的按弦品格位置转换为相对于svg图案上的品格位置
let relativeFret = fretOffset > 0 ? minFret - fretOffset + 1 : minFret;
// 在图案左侧绘制品格位置偏移标记
this.setFretOffset(svg, relativeFret, minFret, barreStringTo === 6);
// 在图案上侧绘制和弦名称
this.setChordName(svg, chordName.getChordName(chordTone));
// 将生成号的svg图案塞到指定结构中
target ? target.appendChild(svg) : document.body.appendChild(svg);
}
}

```

svg也是能使用css修饰部分属性的，公用样式得加上：

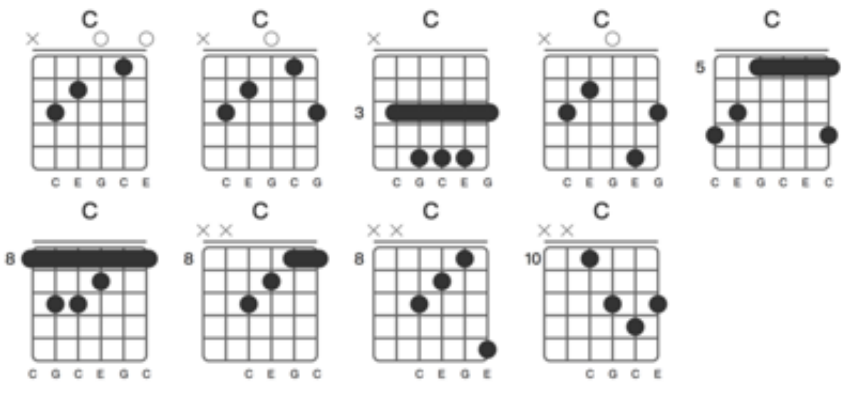
```
.chord-svg{
```



```
width: 200px;
height: 220px;
}
.chord-rect{
width: 100px;
height: 100px;
fill:none;
stroke:#666;
stroke-width:2;
}
.chord-gird{
stroke: #666;
stroke-width: 2;
fill: none;
}
.chord-forbidden,.chord-blank-circle{
stroke: #666;
stroke-width: 1;
fill: none;
}
.chord-block-circle{
fill: #333;
}
.chord-barre{
height: 16px;
fill:#333;
}
.chord-barre-fret{
fill:#333;
font-size: 14px;
font-weight: 700;
}
.chord-string-key{
fill:#333;
font-size: 10px;
font-weight: 700;
}
.chord-name{
fill:#333;
```

```
font-size: 20px;
font-weight: 700;
}
```

甭当当一个运行示例：



```
> let chord = new GuitarChord();
let chordTone = ['1', '3', '5'];
let chordResult = chord.chord(chordTone);
let svg = new ChordSvg();
chordResult.forEach((chordTone) => {
  svg.drawChord(chordTone, chordTone);
});
undefined
```

当然，我怎会止步于此。

基于以上已经实现的代码，我又折腾出了一个网页工具，在数字上左右拖动来改变和弦的组成音，从而时时计算和弦指法图：

[在线试玩地址](#)

2

#4

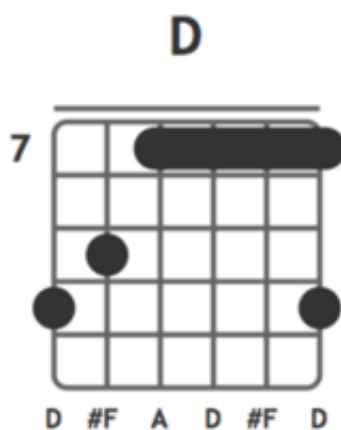
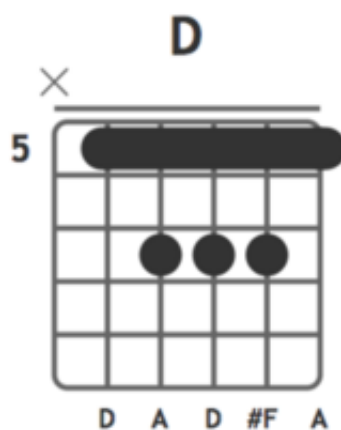
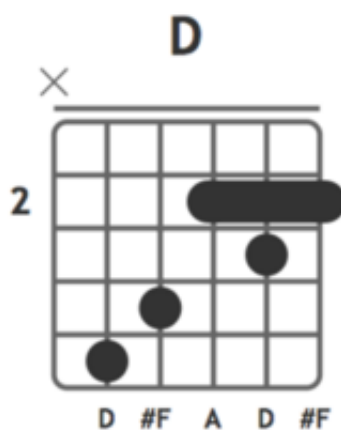
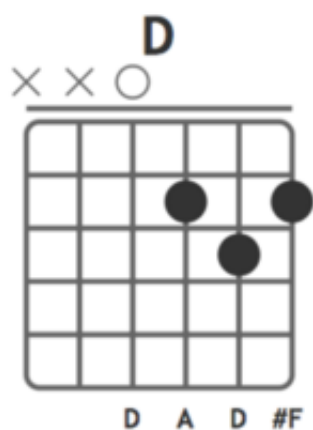
6

左右拖动可改变和弦组成音

三音和弦

四音和弦

试的久了，就会找到规律



4

6

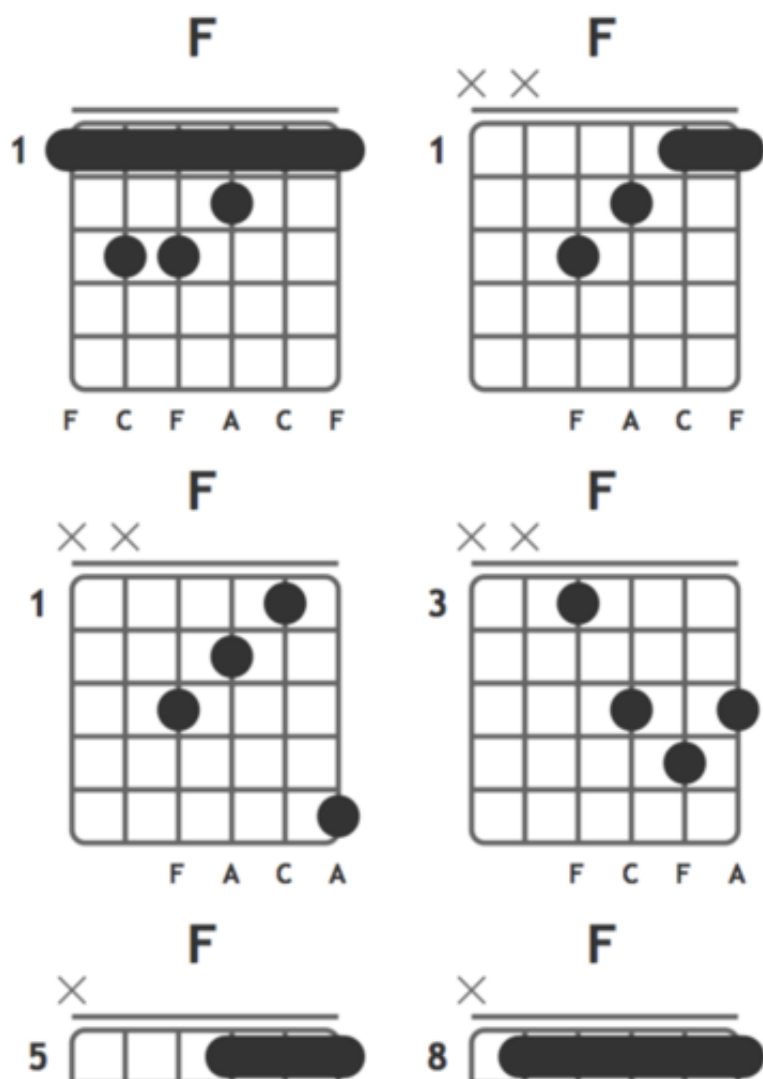
1

左右拖动可改变和弦组成音

三音和弦

四音和弦

你没看错，这是和弦



如果你不按套路出牌，给了间隔古怪的组成音，可能会这样（因为算不出完整的和弦名字了，就

用省略号代替了)：

2                      1                      6

左右拖动可改变和弦组成音

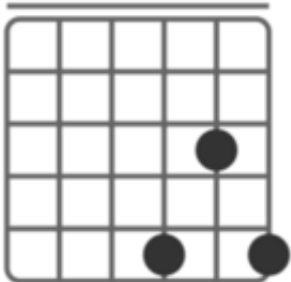
三音和弦

四音和弦

【省略号脸】

D...

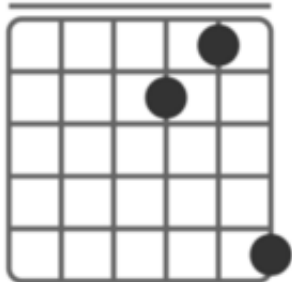
× × ○



D C D A

D...

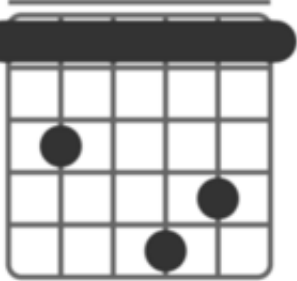
× × ○



D A C A

D...

10



D A C A C D

当然，如果你乱拖一通，大多数情况会是这样：

1

1

b7

左右拖动可改变和弦组成音

三音和弦

四音和弦

组成音太诡异，透露着没图没真相的气息