

小爱开放平台语音技能SDK

- 小爱开放平台语音技能SDK
 - 安装
 - 用法
 - 快速启动
 - HTTPS启动
 - 定义中间件
 - 和KOA结合使用
 - 对接NLU平台
 - API
 - AixBot
 - Constructor
 - use
 - onEvent
 - onIntent
 - onText
 - onRegExp
 - hears
 - onError
 - run
 - httpHandler
 - Context
 - Request
 - Response
 - speak
 - wait
 - query
 - reply
 - directiveAudio
 - directiveTts
 - directiveRecord
 - display
 - setSession
 - playMsgs
 - registerPlayFinishing
 - launchQuickApp
 - launchApp
 - record
 - closeSession
 - notUnderstand
 - body
 - context delegates
 - 其它
 - 作者

[小爱开放平台语音技能](#)的非官方nodejs SDK，帮助你轻松对接小爱开放平台，快速构建起属于自己的语音技能。

使用前需要先在[小爱开放平台](#)注册开发者身份，申请语音技能，并确定服务器URL。具体参见[小爱平台文档](#)。

安装

```
npm install aixbot
```

用法

AixBot和nodejs社区著名的[koa](#)框架用法基本一致，通过定义中间件和事件监听回调来完成任务。

快速启动

以下示例实现了一个简单的语音技能：

- 支持进入和退出技能时的礼貌用语
- 支持用户直接询问"你是谁"
- 其它消息环回播放

```
const AixBot = require('aixbot');

const aixbot = new AixBot();

// define event handler
aixbot.onEvent('enterSkill', (ctx) => {
  ctx.speak('你好').wait();
});

// define text handler
aixbot.hears('你是谁', (ctx) => {
  ctx.speak(`我是Bowen`).wait();
});

// define regex handler, echo message
aixbot.hears(/\W+/, (ctx) => {
  ctx.speak(ctx.request.query);
});

// close session
aixbot.onEvent('quitSkill', (ctx) => {
  ctx.reply('再见').closeSession();
});

// run http server
aixbot.run(8080);
```

HTTPS启动

AixBot默认使用http协议。由于小爱开放平台需要开发者提供https，建议最好在nginx上配置好SSL证书，然后代理到内部aixbot的端口。

AixBot也支持直接以https启动，如下。

```
// config your ssl key and pem
let tlsOptions = {
  key: fs.readFileSync('./keys/1522555444697.key'),
  cert: fs.readFileSync('./keys/1522555444697.pem')
};

aixbot.run(8080, '0.0.0.0', tlsOptions);
```

定义中间件

AixBot支持像koa那样注册中间件。AixBot当前只支持中间件使用`async`和`await`的方式处理异步。

```
const AixBot = require('aixbot');

const aixbot = new AixBot();

// define middleware for response time
aixbot.use(async (ctx, next) => {
  console.log(`process request for '${ctx.request.query}' ...`);
  var start = new Date().getTime();
  await next();
  var execTime = new Date().getTime() - start;
  console.log(`... response in duration ${execTime}ms`);
});

// define middleware for DB
aixbot.use(async (ctx, next) => {
  ctx.db = {
    username : 'Bowen'
  };
  await next();
});

// define event handler
aixbot.onEvent('enterSkill', (ctx) => {
  ctx.speak('你好').wait();
});

// define text handler
aixbot.hears('你是谁', (ctx) => {
  ctx.speak(`我是${ctx.db.username}`).wait();
});
```

```
// define regex handler
aixbot.hears(/\W+/, (ctx) => {
  ctx.speak(ctx.request.query);
});

// close session
aixbot.onEvent('quitSkill', (ctx) => {
  ctx.reply('再见').closeSession();
});

// define error handler
aixbot.onError((err, ctx) => {
  logger.error(`error occurred: ${err}`);
  ctx.reply('内部错误, 稍后再试').closeSession();
});

// run http server
aixbot.run(8080);
```

如上我们定义了两个中间件，一个打印消息的处理时间，一个为context添加访问DB的属性。由于中间件或者消息处理过程中可能会抛出异常，所以我们为异常定义了处理方式`aixbot.onError((err, ctx) => {...})`。

和KOA结合使用

大多数场景下我们只用像上面那样将AixBot独立启动就可以了，但是某些场景下我们需要在同一个程序里同时发布其它的web接口，这时可以将AixBot和koa结合使用。

```
const AixBot = require('aixbot');

const aixbot = new AixBot();

// define aixobot middleware
aixbot.use(async (ctx, next) => {
  ctx.db = {
    username : 'Bowen'
  };
  await next();
});

// define event handler
aixbot.onEvent('enterSkill', (ctx) => {
  ctx.query('你好');
});

// define text handler
aixbot.hears('你是谁', (ctx) => {
  ctx.speak(`我是${ctx.db.username}`).wait();
});

// define regex handler
```

```
aixbot.hears(/\W+/, (ctx) => {
  ctx.speak(ctx.request.query);
});

// close session
aixbot.onEvent('quitSkill', (ctx) => {
  ctx.reply('再见').closeSession();
});

// define error handler
aixbot.onError((err, ctx) => {
  logger.error(`error occurred: ${err}`);
  ctx.reply('内部错误, 稍后再试').closeSession();
});

const Koa = require('koa');
const koaBody = require('koa-body');
const Router = require('koa-router');

const router = new Router();
const app = new Koa();

// koa middleware
app.use(async (ctx, next) => {
  console.log(`process request for '${ctx.request.url}' ...`);
  var start = new Date().getTime();
  await next();
  var execTime = new Date().getTime() - start;
  console.log(`... response in duration ${execTime}ms`);
});

app.use(koaBody());
router.get('/', (ctx, next) => {
  ctx.response.body = 'welcome';
  ctx.response.status = 200;
});

// register aixbot handler to koa router
router.post('/aixbot', aixbot.httpHandler());

app.use(router.routes());

app.listen(8080);
console.log('KOA server is runing...');
```

在上面的例子里，我们没有直接调用[aixbot.run\(\)](#)，而是使用[router.post\('/aixbot', aixbot.httpHandler\(\)\)](#)将aixbot的处理绑定到koa router指定的/aixbot路由上。同时我们为AixBot和koa定义了各自的消息中间件。在运行时会先执行koa的中间件，然后再根据koa的路由规则进行消息分派。分派到/aixbot上的post消息先会执行AixBot的中间件，然后执行对应的已注册的AixBot消息回调。

对接NLU平台

AixBot支持对小爱发来的消息按照事件类型或者消息内容定义回调方法，并支持对消息内容以正则表达式的方式定义规则。但是如果需要完成复杂的语音技能，就必须对接功能完备的NLU处理平台。

对于NLU处理平台，最直接的是使用[小爱开放平台的NLU配置界面](#)进行配置，配置好后在收到的消息里就会携带NLU处理后得到的intent和slot信息。

AixBot可以监听指定的intent，在context中可以取出对应的slot信息。

```
// define intent handler
aixbot.onIntent('query-weather', (ctx) => {
  console.log(JSON.stringify(ctx.request.slotInfo));
});
```

如果需要完成更复杂的NLU处理，可以将AixBot对接其它更专业的NLU处理平台。遗憾的是[DialogFlow](#)、[wit.ai](#)目前都在墙外，微软的[LUIS](#)当前还可以用。国内类似的开放平台也有，基本和小爱当前的NLU能力差不多。作为一名程序员，说实话我不是很喜欢这种通过网页配置的方式来构建对话，我更喜欢经过良好封装的能够以代码的形式来定义和处理对话的chatbot引擎库，这样可以更加灵活地完成复杂功能。如果你自己有类似的NLU处理能力，那就会很方便了。

AixBot和非小爱的NLU平台对接，无非是在AixBot的回调里面将小爱发来消息里的对话内容转发到对应的NLU平台，然后根据NLU平台的返回结果构造给小爱的回复。这里和具体的NLU平台相关，就不再详述了。

API

AixBot

AixBot API reference

```
const AixBot = require('aixbot')
```

Constructor

Initialize new AixBot bot.

```
const aixbot = new AixBot([appId])
```

Param	Type	Description
[appId]	String	app_id of skill

在小爱开放平台上申请的每一个技能都有一个`app_id`。如果需要对收到的每条消息的`app_id`进行严格校验，则在构造AixBot的实例时提供该值。

use

Registers a middleware.

```
aixbot.use(...middleware)
```

Param	Type	Description
middleware	function	Middleware function

```
aixbot.use(async (ctx, next) => {  
  ctx.db = {  
    username : 'Bowen'  
  };  
  await next();  
});
```

onEvent

Registers event handler.

aixbot.onEvent(eventType, handler)

Param	Type	Description
eventType	String	event type
handler	function	handler function

现在支持以下事件类型：

Event Type	Description
enterSkill	进入技能
quitSkill	离开技能
inSkill	技能进行中
noResponse	音箱无响应
recordFinish	录音完成
recordFail	录音失败
playFinishing	录音播放即将完成

```
aixbot.onEvent('enterSkill', (ctx) => {  
  ctx.speak('你好').wait();  
});  
  
aixbot.onEvent('inSkill', (ctx) => {  
  console.log(`received : ${ctx.request.query}`);  
});
```

注意：**inSkill**事件的处理优先级是最低的，比随后介绍的**onIntent**、**onText**和**onRegExp**都要低。可以用它来做一些默认处理。

onIntent

Registers intent handler.

aixbot.onIntent(intent, handler)

Param	Type	Description
intent	String	intent name
handler	function	handler function

```
aixbot.onIntent('query-weather', (ctx) => {  
  console.log(JSON.stringify(ctx.request.slotInfo));  
});
```

onText

Registers text handler.

aixbot.onText(text, handler)

Param	Type	Description
text	String	query content
handler	function	handler function

```
aixbot.onText('hi', (ctx) => {  
  ctx.speak('hello');  
});
```

onRegExp

Registers regex handler.

aixbot.onRegExp(regex, handler)

Param	Type	Description
regex	RegExp	regular expression
handler	function	handler function


```
aixbot.onRegExp(/\d+/, (ctx) => {  
  ctx.speak(`收到数字: ${ctx.request.query}`);  
});
```

注意：所有 **regex handler** 的优先级低于 **text handler**。

hears

Wrapper of onText and onRegExp.

aixbot.hear(text, handler)

Param	Type	Description
text	String or RegExp	query or regular expression
handler	function	handler function

```
aixbot.hears('你是谁', (ctx) => {  
  ctx.speak(`我是${ctx.db.username}`).wait();  
});  
  
aixbot.hears(/\W+/, (ctx) => {  
  ctx.speak(ctx.request.query);  
});
```

onError

Registers error handler.

aixbot.onError(handler)

Param	Type	Description
handler	function	handler function

```
aixbot.onError((err, ctx) => {  
  logger.error(`error occurred: ${err}`);  
  ctx.reply('内部错误, 稍后再试').closeSession();  
});
```

run

Run http/https server.

aixbot.run(port, host, tlsOptions)

Param	Type	Description
port	number	port number
host	String	host address
tlsOptions	object	https options

如果不提供`tlsOptions`，则启动`http server`，否则启动`https server`

```
let tlsOptions = {
  key: fs.readFileSync('./keys/1522555444697.key'),
  cert: fs.readFileSync('./keys/1522555444697.pem')
};

aixbot.run(8080, '0.0.0.0', tlsOptions);
```

httpHandler

get middleware for KOA.

`aixbot.httpHandler()`

```
const router = new Router();
const app = new Koa();
app.use(koaBody());
router.post('/aixbot', aixbot.httpHandler());
app.use(router.routes());
app.listen(8080);
```

Context

Context API reference.

Context是每一个Aixbot中间件和消息回调的参数，通过它可以得到request和response，访问request和response的属性和方法。

```
aixbot.onEvent('enterSkill', (ctx) => {
  console.log(JSON.stringify(ctx.request.body)); // 打印接收消息体的所有内容
  console.log(ctx.request.query); // 打印接收到的消息文本；具体Request封装过的
  属性和接口参见Request的API介绍
  ctx.response.reply('欢迎! '); // 构造回复消息；具体Response封装过的属性和接口
  参见Response的API介绍
  console.log(JSON.stringify(ctx.response.body)); // 打印发送消息体的所有内容
});
```

另外，为了方便使用，Context代理了Response的一些主要接口，这些接口可以通过Context直接使用。例如：

```
aixbot.onEvent('enterSkill', (ctx) => {  
  ctx.reply('欢迎! '); // 效果和 ctx.response.reply('欢迎! ') 相同  
  console.log(JSON.stringify(ctx.body)); // ctx.body 和 ctx.response.body  
  相同  
});
```

由于Response支持连贯接口调用，所以Context上代理的Response接口也同样支持。

```
aixbot.hears('你是谁', (ctx) => {  
  ctx.speak('我是Bowen, 你是谁? ').wait(); // wait()指示开启麦克风，用于直接的多  
  轮对话  
});
```

最后，Context的存在方便中间件为其添加其它的属性和方法：

```
aixbot.use(async (ctx, next) => {  
  ctx.db = {  
    username : 'Bowen'  
  };  
  await next();  
});  
  
aixbot.hears('你是谁', (ctx) => {  
  ctx.speak(`我是${ctx.db.username}`).wait();  
});
```

Request

Request API reference.

Request封装了从小爱收到的消息体。通过Context可以访问到Request实例：`ctx.request`。

Request对接收消息体进行了封装，对常用字段提供了直接的读取属性。

attribute	type	Description
body	object	消息体原始内容
query	String	message.request.query
session	object	message.session
appId	String	message.session.application.app_id
user	object	message.session.user
context	object	message.context

attribute	type	Description
slotInfo	object	message.request.slot_info
intentName	String	message.request.slot_info.intent_name
eventType	String	message.request.event_type
eventProperty	object	message.request.event_property
requestId	String	message.request.request_id
requestType	number	message.request.type
isEnterSkill	boolean	message.request.type == 0
isInSkill	boolean	message.request.type == 1
isQuitSkill	boolean	message.request.type == 2
isNoResponse	boolean	message.request.no_response
isRecordFinish	boolean	message.request.event_type == 'leavemsg.finished'
isRecordFail	boolean	message.request.event_type == 'leavemsg.failed'
isPlayFinishing	boolean	message.request.event_type == 'mediaplayer.playbacknearlyfinished'

```

aixbot.hears(/\W+/, (ctx) => {
  console.log(ctx.request.appId);
  console.log(ctx.request.query);
  if (ctx.request.isNoResponse) {
    console.log('received no response');
  }
  // ...
})

```

Response

Response API reference.

Response封装了发送给小爱的消息，通过`ctx.response`可以获取到Response的实例。

Response对发送消息体进行了封装，提供了更具有语义性的操作接口。

speak

Reply a text.

`ctx.response.speak(text)`

Param	Type	Description
text	String	返回的消息文本

speak默认是关闭麦克风的，如果想要打开麦克风则需要和后面的wait接口一起使用。

wait

Open mic.

```
ctx.response.speak(text).wait()
```

wait接口不能单独使用，必须跟在其它有内容回复的接口后面。

query

response.speak(text).wait()的语法糖，可以直接写 response.query(text)

reply

与response.speak(text)等价，可以直接写 response.reply(text)

directiveAudio

Reply a audio directive.

```
directiveAudio(url, token, offsetMs)
```

Param	Type	Description
url	String	资源url
token	String	获取资源的token
offsetMs	Long	偏移时间

directiveTts

Reply a tts directive.

```
directiveTts(text)
```

Param	Type	Description
text	String	语音合成文本

directiveRecord

Reply a record directive.

```
directiveRecord(fileId)
```

Param	Type	Description
fileId	String	录音文件ID

display

Reply a display.

`display(type, url, text, template)`

Param	Type	Description
type	<code>Int</code>	1: html, 2: native ui, 3: widgets
url	<code>String</code>	html address
text	<code>String</code>	display text
template	<code>UITemplate</code>	参见 UITemplate

setSession

Add paramter in session.

为当前对话上下文的session中添加变量，小爱会在随后的消息中携带该session参数。

`setSession(obj)`

Param	Type	Description
obj	<code>Any</code>	parameter store in session

playMsgs

Reply to play record msgs.

指示播放列表中所有的录音文件。

`playMsgs(fileIdList)`

Param	Type	Description
fileIdList	<code>Array</code>	file_id array

```
ctx.response.speak('请收听录音').playMsgs(['4747c167f000400f15f4d42x'])
```

registerPlayFinishing

指示播放录音即将完成后发送回调消息，具体参见[小爱相关文档](#)

```
ctx.response.speak('请收听录音').playMsgs(['4747c167f000400f15f4d42x']).registerPlayFinishing();
```

launchQuickApp

启动特定路径的快应用。快应用语音技能的注册及配置见[小爱文档](#)。

launchQuickApp(path)

Param	Type	Description
path	String	path of quick app

```
ctx.response.launchQuickApp('/')
```

launchApp

启动APP。启动APP的语音技能的注册及配置见[小爱文档](#)。

launchApp(type, uri, permission)

Param	Type	Description
type	String	启动APP的intent的类型；支持的类型 1 activity； 2 service； 3 broadcast
uri	String	启动APP的路径
permission	String	权限信息；非必须参数

```
ctx.response.launchApp('activity', 'xxxxxxx')
```

record

指示开始录音，跟在回复后面使用。

```
ctx.response.speak('start record').record()
```

closeSession

指示结束回话，跟在回复后面使用。

```
ctx.response.speak('bye').closeSession()
```

notUnderstand

指示未理解的对话，跟在回复后面使用。

```
ctx.response.speak('what').notUnderstand()
```

body

获取消息体内容

```
ctx.response.speak('hello');  
console.log(JSON.stringify(ctx.response.body));
```

context delegates

为了方便使用，Context对Response的下列属性和方法进行了代理：

- `speak`
- `reply`
- `query`
- `directiveAudio`
- `directiveTts`
- `directiveRecord`
- `display`
- `playMsgs`
- `launchQuickApp`
- `launchApp`
- `body`

```
ctx.speak('hi').wait(); // same as : ctx.response.speak('hi').wait()
```

其它

源码在[github](#)，有问题请提issue。

使用 `npm test`可以对源码进行测试。

如果运行时想打开AixBot的debug打印，可以在启动时加上 `DEBUG=aixbot:*`，例如`DEBUG=aixbot:* node index.js`。

本人使用的是 `node 8.11.1`版本，其它更低版本的不支持`class, const, let, async, await`等特性的node版本请绕路。

作者

- Bowen
- Email: e.wangbo@gmail.com