Rasa 对话引擎

注意

这是 Rasa Core 0.8.5 版的文档。确保为本地安装选择适当的文档版本！

欢迎来到 Rasa 文档！

我在看什么？

Rasa 是构建会话式软件的框架：Messenger / Slack 僵尸程序，Alexa 技能等。我们将 在此文档中将其缩写为 **bot**。您可以

- 实现您的机器人可以在 Python 代码中执行的操作（推荐），
- 或者使用 Rasa Core 作为 web 服务（实验性，请参阅 HTTP 服务器）。

它有什么好处？

`if/else` 您的机器人的逻辑不是基于一堆陈述，而是基于训练有关示例对话的概率模型。
这听起来比写几句话更难

在一个项目的开始阶段，只是硬编码一些逻辑确实比较容易。当你想要通过它时，Rasa 会帮助你，创建一个可以处理更多复杂问题的机器人。
我可以看到它在行动吗？

我们认为你永远不会问！确保遵循安装， 然后检查构建一个简单的机器人！

入门

- 动机
- 但我不用 python 代码！
- 安装

教程

- 建立一个简单的机器人
- 监督学习教程
- 互动学习
- 没有 Python 的 Rasa Core

深潜

- 域，插槽和操作
- 故事 - 训练数据
- 常见模式
- 探究 - 如何融合在一起
- HTTP 服务器
- 连接到消息和语音平台
- 计划提醒

Python API

- 代理人
- 事件

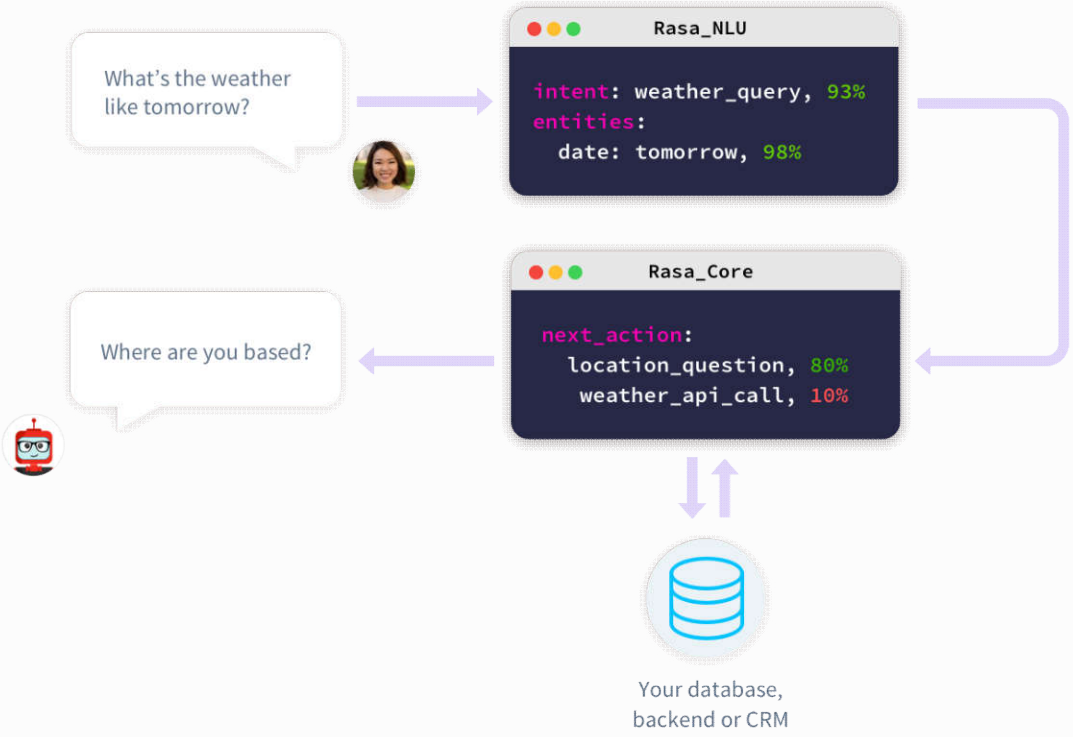
开发者文档

- Featurization
- 解释器
- 自定义策略
- 跟踪对话状态
- 迁移指南
- 更改日志

动机

Rasa Core 适用于哪里

Rasa Core 接受结构化输入：**意图**和**实体**， **按钮点击**等，并决定你的机器人下一步应该运行哪个**动作**。如果您希望系统处理自由文本，则还需要使用 Rasa NLU 或其他 NLU 工具。



指导原则

Rasa Core 的主要思想是将对话视为流程图不会扩展。很难明确地推断出*所有可能的对话*，但如果对方是对还是错，在对话中间很容易分辨出来。

错误的方式™

实现对话流的典型方法是使用状态机。例如，你可能需要从用户收集一些数据来完成他们的订单，然后手动把他们经历的状态 `BROWSING`， `CHECKING_OUT`， `ADDING_PAYMENT`， `ORDER_COMPLETE`， 等的复杂性来自于当用户从“幸福路径”流浪，你需要添加代码用于处理问题，澄清，比较，拒绝等。将这些边缘案例手动插入状态机是一个很大的麻烦。

一个典型的“简单”机器人可能会有 5-10 个州和几百个规则来管理它的行为。当你的机器人不像你想要的那样工作时，找出错误是非常棘手的。

同样，当你想添加一些新功能时，你最终会碰到你之前编写的规则，并且变得越来越难。

我们的观点是，从字面上理解流程图并将它们实现为状态机并不是一个好主意，但**流程图对于描述开心路径和可视对话仍然有用**。

Rasa 方式

`if/else`Rasa 机器人并没有编写大量的语句，而是从真实的对话中学习。概率模型选择要采取的行动，并且可以使用监督，强化或交互式学习来训练。

这种方法的优点是：

- 调试更容易
- 你的机器人可以更灵活
- 无需编写更多代码，您的机器人可以从经验中改进
- 您可以在不调试数百条规则的情况下为您的机器人添加新功能。

从哪儿开始

在完成安装之后，大多数用户应该从 构建简单的机器人开始。但是，如果您已经有一系列想要用作训练集的对话，请查看监督学习教程。

问题

为什么选择 python？

由于其机器学习工具的生态系统。转到但我不使用 Python 代码！了解详情。
这仅适用于 ML 专家吗？

如果你对机器学习一无所知，你可以使用 Rasa，但如果你这么做，那么实验起来很容易。
我需要多少训练数据？

通过使用交互式学习，您可以从零训练数据中引导。试试教程！

但我不用 **python** 代码！

虽然 python 是机器学习的*通用语*，但我们知道大多数聊天机器人都是用 javascript 编写的，而且很多企业在 Java，C＃等平台上构建和发布应用程序更加舒适。

即使您不想使用 python，我们也尽一切努力确保您**仍然**可以使用 Rasa Core。但是，请考虑 Rasa Core 是一个*框架*，并且不像 Rasa NLU 那样容易适合 REST API。

Rasa Core 与最小的 Python

您可以通过以下方式与 Rasa Core 构建聊天机器人：

- 定义一个域（一个 yaml 文件）
- 写作/收集故事（markdown 格式）
- 运行 python 脚本来训练和运行你的机器人

您需要编写 python 的唯一部分是当您要定义自定义操作时。有一个名为 requests 的优秀 python 库，这使得 HTTP 编程变得轻松无比。如果 Rasa 只需要通过 HTTP 与您的其他服务进行交互，您的操作将全部如下所示：

```
from rasa_core.actions import Action
import requests

class ApiAction(Action):
    def name(self):
        return "my_api_action"

    def run(self, dispatcher, tracker, domain):
        data = requests.get(url).json
        return [SlotSet("api_result", data)]
```

带有 ZERO Python 的 Rasa Core

如果你真的被限制为不使用任何 python，你也可以通过 HTTP API 使用 Rasa Core 。

安装

安装 Rasa Core 以开始使用 Rasa 堆栈。

安装 Rasa Core

推荐的安装 Rasa Core 的方法是使用 pip：

```
pip install rasa_core
```

除非您已安装 numpy&scipy，否则我们强烈建议您安装并使用 Anaconda。

如果您想使用 Rasa 的最新版本，请使用 github + setup.py：

```
git clone https://github.com/RasaHQ/rasa_core.git
cd rasa_core
pip install -r requirements.txt
pip install -e .
```

注意

如果您想更改 Rasa Core 代码并想运行测试或构建文档，则需要安装开发依赖项：

```
pip install -r dev-requirements.txt
pip install -e .
```

添加自然语言理解

我们使用 Rasa NLU 进行意图分类和实体提取。为了得到它，运行

```
pip install rasa_nlu
```

完整的说明可以在这里找到。

您也可以使用其他 NLU 服务，如 wit.ai，dialogflow 或 LUIS。实际上，如果您的消息传递应用程序使用按钮而不是自由文本，则根本不需要使用 NLU。

入门

要查看刚安装的 Rasa Core，请参阅建立简单 Bot 的入门教程。

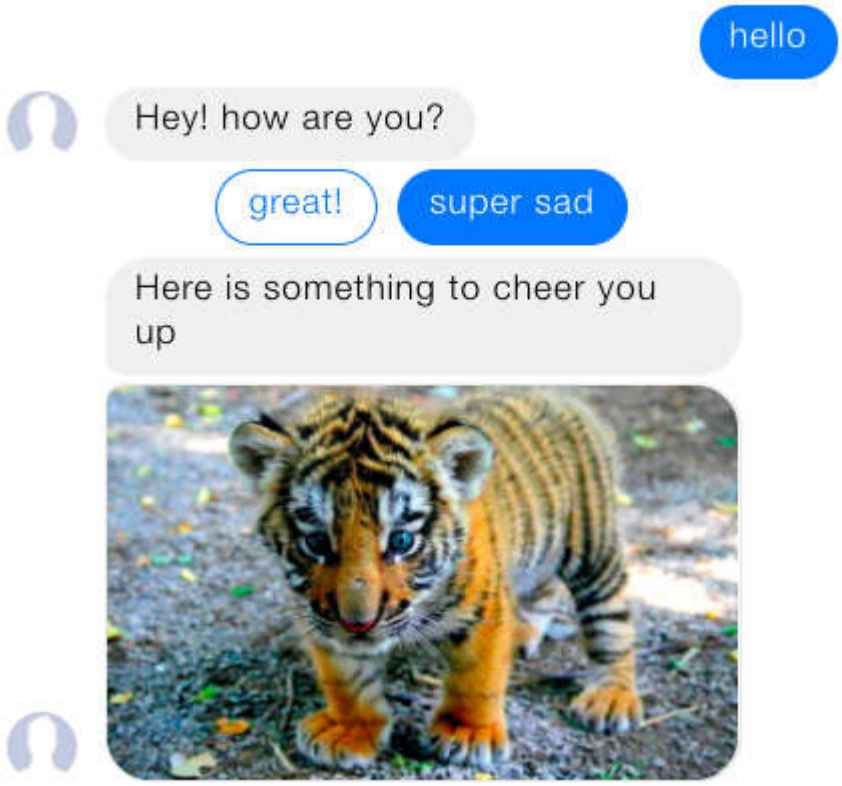
建立一个简单的机器人

注意

本教程将向您展示构建 bot 的不同部分。请注意，这是一个快速入门的小例子。它不包含大量的训练数据，所以最终的机器人性能有一定的提升空间。

GitHub 上的示例代码

在这里我们展示了如何创建你的第一个机器人，添加一个 Rasa 应用程序的所有部分。如果你也看看探究 - 这一切如何融合在一起，这可能会更容易遵循。



目标

我们将创建一个非常简单的机器人，检查我们目前的心情，如果我们感到难过，就试图让我们振作起来。它会查询我们的情绪，并根据我们的回复将回应一个有趣的图像或消息。

我们首先创建一个项目文件夹：

```
mkdir moodbot &&  cd moodbot
```

我们需要创建两个数据文件（对话故事和 NLU 示例）以及两个配置文件（对话域和 NLU 配置）。最终的结构应该如下所示：

```
moodbot/
├── data/
│   ├── stories.md          # dialogue training data
│   └── nlu.md              # nlu training data
├── domain.yml              # dialogue configuration
└── nlu_model_config.json   # nlu configuration
```

让我们来看看它们中的每一个！

1.定义一个域

我们首先需要的是一个 `Domain`。该域定义了你的机器人居住的宇宙。

以下是我们情绪机器人的一个示例域 `domain.yml`：

```
intents:
  - greet
  - goodbye
  - mood_affirm
  - mood_deny
  - mood_great
  - mood_unhappy

actions:
- utter_greet
- utter_cheer_up
- utter_did_that_help
- utter_happy
- utter_goodbye

templates:
  utter_greet:
  - text: "Hey! How are you?"
    buttons:
    - title: "great"
      payload: "great"
    - title: "super sad"
      payload: "super sad"

  utter_cheer_up:
  - text: "Here is something to cheer you up:"
    image: "https://i.imgur.com/nGF1K8f.jpg"

```

```
29  utter_did_that_help:
30  - text: "Did that help you?"
31
32  utter_happy:
33  - text: "Great carry on!"
34
35  utter_goodbye:
36  - text: "Bye"
```

那么不同的部分是什么意思？

`intents`	你期望用户说的东西。有关详细信息，请参阅 Rasa NLU。
`entities`	您想要从消息中提取的信息片段。有关详细信息，请参阅 Rasa NLU。
`actions`	你的机器人可以做和说的东西
`slots`	在会话期间跟踪的信息（例如用户年龄）
`templates`	你的机器人可以说的东西的模板字符串

在我们的简单的例子，我们不需要 `slots` 和 `entities`，所以这部分没有在我们的定义出现。

这是如何融合在一起的？ Rasa Core 取 `intent`， `entities` 以及内部的对话状态，并且选择的所述一个 `actions` 应该下一个执行。如果该动作只是向用户说明，Rasa 将在域中查找匹配模板（动作名称等于完全模板， `utter_greet` 如上例所示），填写所有变量并进行响应。对于不仅仅发送消息的操作，您可以将它们定义为 python 类，并通过它们的模块路径在域中引用它们。有关自定义操作的更多信息，请参阅 定义自定义操作。

注意

还有一个额外的特殊行为， `ActionListen` 意味着在用户说别的之前停止采取进一步行动。它没有在中指定 `domain.yml`

2.定义一个解释器

解释器负责解析消息。它执行自然语言理解（NLU）并将消息转换为结构化输出。在这个例子中，我们将使用 Rasa NLU 来达到这个目的。

在 Rasa NLU 中，我们需要定义我们的机器人应该能够以 Rasa NLU 训练数据格式处理的用户消息。在本教程中，我们将使用 Markdown 格式来获取 NLU 训练数据。我们来创建一些意图示例 `data/nlu.md`：

```
## intent:greet
- hey
- hello
- hi
- hello there
- good morning
- good evening
- moin
- hey there
- let's go
- hey dude
- goodmorning
- goodevening
- good afternoon

## intent:goodbye
- cu
- good by
- cee you later
- good night
- good afternoon
- bye
- goodbye
- have a nice day
- see you around
- bye bye
- see you later

## intent:mood_affirm
- yes
- indeed
- of course
- that sounds good
- correct

## intent:mood_deny
- no
- never
- I don't think so
- don't like that
```

```
- no way

## intent:mood_great
- perfect
- very good
- great
- amazing
- feeling like a king
- wonderful
- I am feeling very good
- I am great
- I am amazing
- I am going to save the world
- super
- extremely good
- so so perfect
- so good
- so perfect

## intent:mood_unhappy
- my day was horrible
- I am sad
- I don't feel very well
- I am disappointed
- super sad
- I'm so sad
- sad
- very sad
- unhappy
- not so good
- not very good
- extremly sad
- so saad
- so sad
```

此外，我们需要一个配置文件 `nlu_model_config.json`，对于 NLU 模型：

```
{
  "pipeline": "spacy_sklearn",
  "path" : "./models/nlu",
  "data" : "./data/nlu.md"
}
```

我们现在可以使用我们的示例来训练 NLU 模型（确保首先 安装 Rasa NLU 以及 spaCy）。

让我们跑吧

```
python -m rasa_nlu.train -c nlu_model_config.json --fixed_model_name current
```

训练我们的 NLU 模型。`models/nlu/default/current` 应创建一个包含 NLU 模型的新目录。请注意，这 `default` 表示项目名称，因为我们没有明确指定它 `nlu_model_config.json`。

注意

要收集关于上述配置的更多见解，并且 Rasa NLU 功能将转入 Rasa NLU 文档。

3.定义故事

到目前为止，我们已经有了一个 NLU 模型，一个定义我们的机器人可以采取的动作的域以及它应该处理的输入（意图和实体）。我们仍然错过了中心部分，**故事告诉我们的机器人在对话的哪一点做什么**。

一个**故事**是，对话系统训练数据样本。有两种不同的方式来创建故事（你可以混合它们）：

- 手动创建故事，直接将它们写入文件
- 使用交互式学习创建故事（请参阅交互式学习）。

对于这个例子，我们将通过直接写入故事来创建故事 `stories.md`。故事以`##`一个字符串作为标识符开始。用户操作以星号开头，机器人操作由以短划线开头的行指定。故事的结尾用换行符表示。有关数据格式的更多信息，请参阅故事 - 训练数据。

说够了，让我们来看看我们的故事：

```
1 ## happy path              <!-- name of the story - just for debugging -->
 2* greet
 3   - utter_greet
 4* mood_great              <!-- user utterance, in format _intent[entities] -->
 5   - utter_happy
 6
 7## sad path 1              <!-- this is already the start of the next story -->
 8* greet
 9   - utter_greet           <!-- action of the bot to execute -->
10* mood_unhappy
11   - utter_cheer_up
```

```
12  - utter_did_that_help
13* mood_affirm
14  - utter_happy
15
16## sad path 2
17* greet
18  - utter_greet
19* mood_unhappy
20  - utter_cheer_up
21  - utter_did_that_help
22* mood_deny
23  - utter_goodbye
24
25## say goodbye
26* goodbye
27  - utter_goodbye
```

请注意，虽然直接用手写故事比使用交互式学习要快得多，但在使用插槽时需要特别小心，因为它们需要在故事中正确设置。

把碎片放在一起

我们仍然需要做两件事：训练对话模型并运行它。

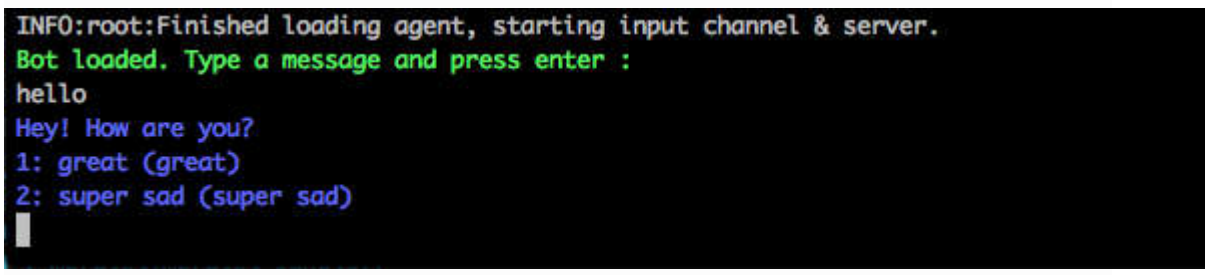
要训　　练对话模型，请运行：

```
python -m rasa_core.train -s data/stories.md -d domain.yml -o models/dialogue --epochs 300
```

这将训练 `300` 时代的对话模型并将其存储到 `models/dialogue`。现在我们可以使用训练好的对话模型和之前创建的 NLU 模型来运行我们的机器人。在这里，我们只需要在命令行中与 bot 进行交谈：

```
python -m rasa_core.run -d models/dialogue -u models/nlu/default/current
```

我们终于得到它了！包含 Rasa Core 所有重要部分的最小机器人。



注意

按钮模拟在控制台输出中不起作用，您需要输入“great”或“sad”而不是数字 1 或 2。

奖励：处理来自 Facebook 的消息

如果要处理来自 Facebook 而不是命令行的输入，则可以在创建包含连接到 Facebook 的信息的凭证文件后，将其指定为运行命令的一部分。让我们把它放到 `fb_credentials.yml`：

```
verify: "rasa-bot"
secret: "3e34709d01ea89032asdebfe5a74518"
page-access-token: "EAAbHPa7H9rEBAAuFk4Q3gPKbDedQnx4djJJ1JmQ7CAqO4iJKrQcNT0wtD"
```

如果您对 Facebook Messenger 机器人不熟悉，请转到 Facebook Messenger 安装程序以获取对不同值的解释。

设置好之后，我们现在可以运行机器人

```
python -m rasa_core.run -d models/dialogue -u models/nlu/current \
   --port 5002 --connector facebook --credentials fb_credentials.yml
```

现在它将处理用户发送到 Facebook 页面的消息。

监督学习教程

注意

本教程将介绍如何直接从 python 使用 Rasa Core。我们将深入探讨图书馆的不同概念和整体结构。你应该已经熟悉术语，故事，并且具有 NLU 的一些知识（如果不是，首先要建立一个简单的机器人）。在这里，我们将使用 GitHub 上的示例代码。

目标

在这个例子中，我们将通过在示例对话上训练神经网络来创建餐厅搜索机器人。用户可以使用接近的东西与机器人联系，机器人会询问更多细节，直到准备建议一家餐馆。`"I want a mexican restaurant!"`

第一步

我们先来看看我们的餐厅机器人目录。所有示例代码片断都假定您正在运行该项目目录中的代码：

```
cd examples/restaurantbot
```

1.域

我们来看看域定义 `restaurant_domain.yml`：

除前面的示例外，还有两个部分：`slots` 和 `entities`。

`slots` 用于存储用户偏好，如餐厅的美食和价格范围。 `entities` 与插槽密切相关。插槽随时间而更新，实体是从用户消息中获取的原始信息。但是插槽也可用于存储有关外部世界的信息，例如 API 调用的结果或从数据库读取的用户配置文件。这里我们有一个插槽叫做 `matches` 哪个存储由 API 返回的匹配餐厅。

```
slots:
  cuisine:
    type: text
  people:
    type: text
  location:
    type: text
  price:
    type: text
  info:
    type: text
  matches:
    type: unfeaturized

entities:
- location
- info
- people
- price
- cuisine

intents:
 - greet
 - affirm
 - deny
```

```
- inform
- thankyou
- request_info

templates:
  utter_greet:
    - "hey there!"
  utter_goodbye:
    - "goodbye :("
    - "Bye-bye"
  utter_default:
    - "default message"
  utter_ack_dosearch:
    - "ok let me see what I can find"
  utter_ack_findalternatives:
    - "ok let me see what else there is"
  utter_ack_makereservation:
    - text: "ok making a reservation"
      buttons:
        - title: "thank you"
          payload: "thank you"
  utter_ask_cuisine:
    - "what kind of cuisine would you like?"
  utter_ask_howcanhelp:
    - "how can I help you?"
  utter_ask_location:
    - "where?"
  utter_ask_moreupdates:
    - "if you'd like to modify anything else, please tell me what"
  utter_ask_numpeople:
    - "for how many people?"
  utter_ask_price:
    - text: "in which price range?"
      buttons:
      - title: "cheap"
        payload: "cheap"
      - title: "expensive"
        payload: "expensive"
  utter_on_it:
    - "I'm on it"

actions:
- utter_greet
- utter_goodbye
```

```
- utter_default
- utter_ack_dosearch
- utter_ack_findalternatives
- utter_ack_makereservation
- utter_ask_cuisine
- utter_ask_howcanhelp
- utter_ask_location
- utter_ask_moreupdates
- utter_ask_numpeople
- utter_ask_price
- utter_on_it
- bot.ActionSearchRestaurants
- bot.ActionSuggest
```

自定义操作

在这个例子中，我们也有自定义操作：`bot.ActionSearchRestaurants`和`bot.ActionSuggest`，其中`bot.`代表定义此操作的模块的名称。一个动作可以做的不仅仅是发送消息。以下是调用 API 的自定义操作的一个小例子。请注意，该`run`方法可以使用存储在跟踪器中的插槽的值。

```
class ActionSearchRestaurants(Action):
    def name(self):
        return 'action_search_restaurants'

    def run(self, dispatcher, tracker, domain):
        dispatcher.utter_message("looking for restaurants")
        restaurant_api = RestaurantAPI()
        restaurants = restaurant_api.search(tracker.get_slot("cuisine"))
        return [SlotSet("matches", restaurants)]
```

但是一个领域本身并不是一个机器人; 我们需要一些训练数据来告诉机器人应该在对话的哪个点执行哪些动作。我们需要一些交谈训练数据 - *故事*!

2.训练数据

看一下`data/babi_stories.md`，餐馆机器人的训练对话是在哪里定义的。一个示例故事如下所示：

```
## story_00914561
* greet
```

```
 3 - utter_ask_howcanhelp
 4 * inform{"cuisine": "italian"}
 5 - utter_on_it
 6 - utter_ask_location
 7 * inform{"location": "paris"}
 8 - utter_ask_numpeople
 9 * inform{"people": "six"}
10 - utter_ask_price
11 ...
```

请参阅下面的训练数据以获取有关此训练数据的更多信息。

3.训练你的机器人

只需几个步骤，我们就可以直接从数据到机器人：

1. 训练 Rasa NLU 模型以提取意图和实体。在 NLU 文档中了解更多。
2. 训练对话策略，学会选择正确的行动。
3. 建立一个既有模型 1（NLU）又有模型 2（对话）一起工作的代理，直接从**用户输入**到**行动**。

我们将逐一完成这些步骤。

NLU 模型

为了训练我们的 Rasa NLU 模型，我们需要一个配置文件，你可以在这里找到 `nlu_model_config.json`：

```
1 {
 2  "pipeline": [
 3    "nlp_spacy",
 4    "tokenizer_spacy",
 5    "intent_featurizer_spacy",
 6    "intent_classifier_sklearn",
 7    "ner_crf",
 8    "ner_synonyms"
 9  ],
10  "path" : "./models/nlu",
11  "data" : "./data/franken_data.json"
12 }
```

并且训练数据 `franken_data.json`（详情请参阅 < https://nlu.rasa.ai/dataformat.html >`_）。

我们可以使用训练 NLU 模型

```
python -m rasa_nlu.train -c nlu_model_config.json --fixed_model_name current
```

或使用 Python 代码

```
def train_nlu():
    from rasa_nlu.converters import load_data
    from rasa_nlu.config import RasaNLUConfig
    from rasa_nlu.model import Trainer

    training_data = load_data('data/franken_data.json')
    trainer = Trainer(RasaNLUConfig("nlu_model_config.json"))
    trainer.train(training_data)
    model_directory = trainer.persist('models/nlu/', fixed_model_name="current")

    return model_directory
```

并运行

```
python bot.py train-nlu
```

2014 年 MacBook Pro 上的 NLU 训练约需 18 秒。

定制对话策略

现在我们的机器人需要了解如何回应用户消息。我们通过训练一个或多个 Rasa 核心策略来做到这一点。

对于这个机器人，我们提出了自己的策略。该策略扩展了 Keras 策略，修改了底层神经网络的 ML 架构。查看该 `RestaurantPolicy` 课程 `bot.py` 的血腥详细信息：

```
class RestaurantPolicy(KerasPolicy):
    def model_architecture(self, num_features, num_actions, max_history_len):
        """Build a Keras model and return a compiled model."""
        from keras.layers import LSTM, Activation, Masking, Dense
        from keras.models import Sequential

        n_hidden = 32  # size of hidden layer in LSTM
        # Build Model
```

```
        batch_shape = (None, max_history_len, num_features)

        model = Sequential()
        model.add(Masking(-1, batch_input_shape=batch_shape))
        model.add(LSTM(n_hidden, batch_input_shape=batch_shape))
        model.add(Dense(input_dim=n_hidden, output_dim=num_actions))
        model.add(Activation('softmax'))

        model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
                      metrics=['accuracy'])

        logger.debug(model.summary())
        return model
```

参数 `max_history_len` 和 `n_hidden` 可以改变依赖于任务的复杂性和数据的一个具有的量。`max_history_len` 非常重要，因为它是网络可以进行分类的先前故事步骤的数量。

因为我们已经创建了自定义策略，所以我们无法通过 `rasa_core.train` 像构建简单 Bot 那样运行来训练机器人。该 `bot.py` 脚本显示了如何训练使用自定义策略和操作的机器人。

注意

请记住，您不需要创建自己的策略。使用 *记忆*策略和 *Keras* 策略的默认策略设置工作得很好。不过，您可以随时根据自己的用例对它们进行微调。阅读 探究 - 如何将它们放在一起以获取更多信息。

现在我们训练它：

```
def train_dialogue(domain_file="restaurant_domain.yml",
                   model_path="models/dialogue",
                   training_data_file="data/babi_stories.md"):
    agent = Agent(domain_file,
                  policies=[MemoizationPolicy(), RestaurantPolicy()])

    agent.train(
            training_data_file,
            max_history=3,
            epochs=400,
            batch_size=100,
            validation_split=0.2
    )
```

```
14
15      agent.persist(model_path)
16      return agent
```

此代码创建要训练的策略，并使用故事训练数据来训练并坚持（存储）模型。训练有素的策略的目标是根据 bot 的当前状态来预测下一个动作。

从命令行训练对话策略，运行

```
python bot.py train-dialogue
```

2014 年的 MacBook Pro 训练对话模型大约需要 12 分钟

4.使用机器人

现在我们要将一些碎片粘合在一起来创建一个真正的机器人。我们实例化一个 `Agent`，它拥有我们训练有素的 `Policy`，`Domain` 来自 `models/dialogue` 我们的 NLU 和 `Interpreter` 来自我们的 NLU `models/nlu/default/current`。

对于这个演示，我们将直接发送消息到 Python 控制台的 bot。通过查看连接到消息和语音平台，您可以查看如何构建命令行机器人和 Facebook 机器人。

```
from rasa_core.interpreter import RasaNLUInterpreter
from rasa_core.agent import Agent

agent = Agent.load("models/dialogue",
interpreter=RasaNLUInterpreter("models/nlu/default/current"))
```

然后我们可以尝试发送一条消息：

```
>>> agent.handle_message("/greet")
[u'hey there!']
```

我们终于得到它了！包含 Rasa Core 所有重要部分的最小机器人。

注意

在这里，我们直接提供了潜在的意图 `greet`（参见 固定意图和实体输入），从而忽略了 NLU 对我们信息的解释。
如果您想要处理来自命令行（或另一个输入通道）的输入，则需要处理该通道而不是直接处理消息，例如：

```
from rasa_core.channels.console import ConsoleInputChannel
agent.handle_channel(ConsoleInputChannel())
```

在这种情况下，消息将从命令行中检索，因为我们指定了 `ConsoleInputChannel`。响应也被打印到命令行。

您可以找到一个完整的例子，介绍如何在餐厅机器人的 `bot.py` `run` 方法的命令行中加载代理并与其进行聊天。要从命令行运行机器人，请运行

```
python bot.py run
```

如果机器人看起来卡住或错误地回答，不要担心。所提供的数据集不够多样，无法处理所有可能的输入，正如从下面的可视化训练数据中可以看出的那样。可以使用交互式学习来增加具有自定义故事的训练数据。

细节

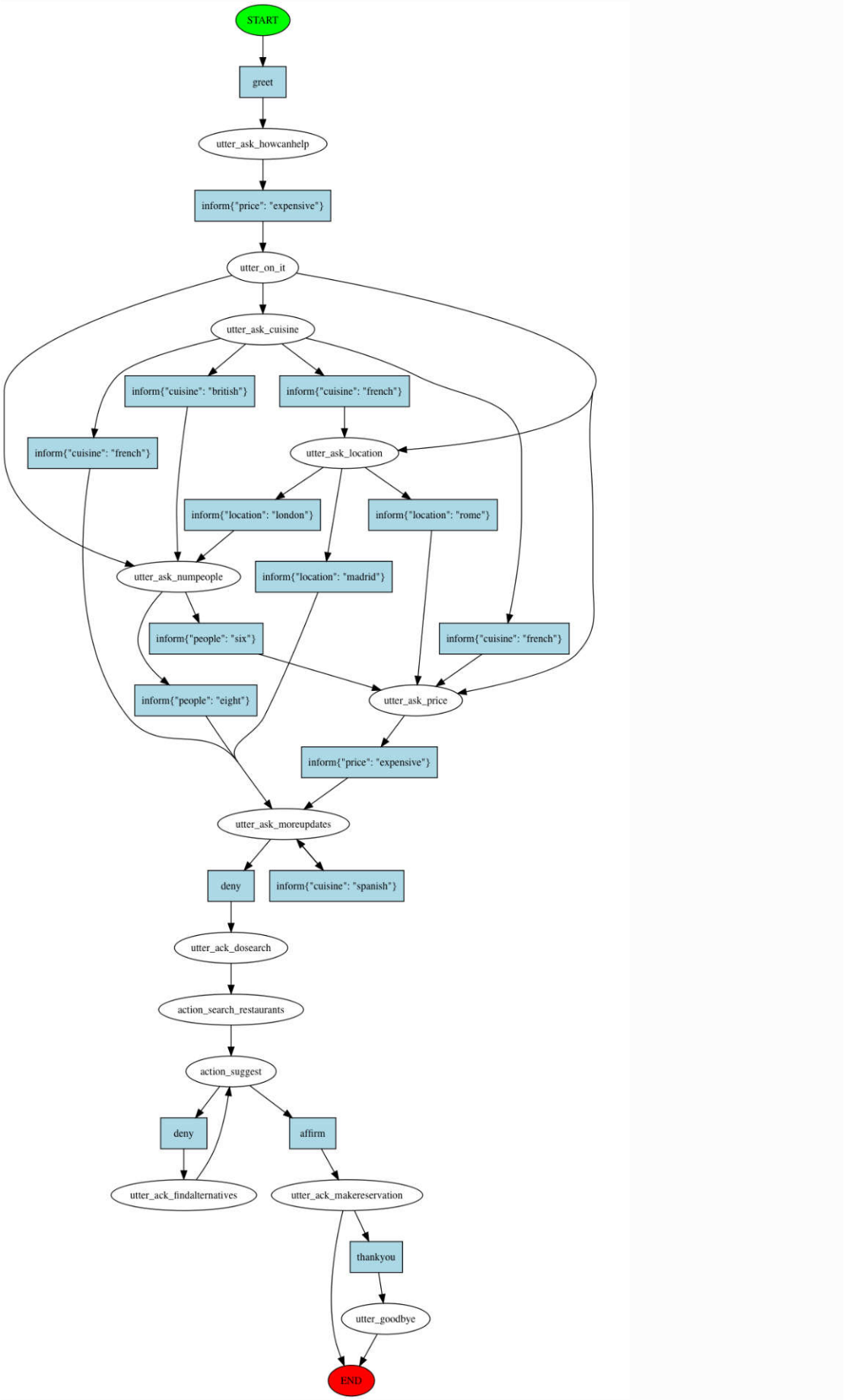
训练数据

训练对话来自 bAbI 对话任务。然而，这些对话中的消息是机器生成的，因此我们将使用来自 DSTC 数据集的真实用户消息来扩充此数据集。幸运的是，这个数据集也在餐厅领域。

注意

bAbI 数据集是机器生成的，在那里有很多对话。训练集中有 1000 个故事，但你并不需要那么多来构建一个有用的机器人。您需要多少数据取决于您定义的操作数量以及要支持的边缘案例数量。但是，几十个故事是一个开始的好地方。
我们已将 bAbI 对话训练集转换为 Rasa 故事格式，您可以从 GitHub 下载故事训练数据。该文件存储在 `data/babi_stories.md`。

查看故事 - 训练数据以获取有关 Rasa Core 数据格式的更多信息。我们还可以将该训练数据可视化以生成类似于流程图的图形：


START
greet
utter_ask_howcanhelp
inform{"price": "expensive"}
utter_on_it
utter_ask_cuisine
inform{"cuisine": "british"}
inform{"cuisine": "french"}
inform{"cuisine": "french"}
utter_ask_location
inform{"location": "london"}
inform{"location": "rome"}
utter_ask_numpeople
inform{"location": "madrid"}
inform{"people": "six"}
inform{"cuisine": "french"}
inform{"people": "eight"}
utter_ask_price
inform{"price": "expensive"}
utter_ask_moreupdates
deny
inform{"cuisine": "spanish"}
utter_ack_dosearch
action_search_restaurants
action_suggest
deny
affirm
utter_ack_findalternatives
utter_ack_makereservation
thankyou
utter_goodbye
END

该图显示了训练数据中执行的所有操作以及它们之间发生的用户消息（如果有的话）。正如你所看到的，流程图很快就会变得复杂。不过，它们可以成为调试 bot 的有用工具。例如，可以清楚地看到没有足够的数据来处理预订。要了解如何构建此图表，请转到 故事可视化。

互动学习

注意

如果你有一个机器人的好主意，但是你没有任何对话可以用作训练数据，那么这是开始的地方。我们假设你已经想过你需要什么意图和实体（ 如果你不知道这些文件是什么的话，请查看 Rasa NLU 文档）。

我们在 GitHub 上使用这个示例代码。

问题

你的机器人通常有明确的目标，它应该与用户交谈时达到。在达到最后阶段之前，对话通常会有许多不同的方式。我们将教你如何使用 Rasa Core 来从最少的训练数据引导全面的对话。

机器人

假设你想建立一个推荐音乐会的机器人。有一个目标：你知道在对话结束时你希望你的机器人提出建议。我们将展示如何在不编写流程图的情况下实现上下文感知行为。例如，如果我们的用户问这个问题： *哪些人有更好的评论？* ，我们的机器人应该知道他们是否想比较*音乐家*或*场地*。

转到 `examples/concertbot` 这个例子。我们走吧！

域名

我们将保持演唱会域名简单，并且不会添加任何插槽。我们也只会支持这些意图： 。这是域定义（）：

`"greet", "thankyou", "goodbye", "search_concerts", "search_venues", "compare_reviews"concert_domain.yml`

```
slots:
  concerts:
    type: list
  venues:
    type: list

intents:
```

```
 8 - greet
 9 - thankyou
10 - goodbye
11 - search_concerts
12 - search_venues
13 - compare_reviews
14
15 entities:
16 - name
17
18 templates:
19   utter_greet:
20     - "hey there!"
21   utter_goodbye:
22     - "goodbye :("
23   utter_default:
24     - "default message"
25   utter_youarewelcome:
26     - "you're very welcome"
27
28 actions:
29   - utter_default
30   - utter_greet
31   - utter_goodbye
32   - utter_youarewelcome
33   - actions.ActionSearchConcerts
34   - actions.ActionSearchVenues
35   - actions.ActionShowConcertReviews
36   - actions.ActionShowVenueReviews
```

无状态的故事

我们首先在 Rasa 故事格式的一些简单对话上训练一个无状态模型。这意味着我们用一个用户话语定义对话，并且只有少数（通常是一个）机器人动作作为响应。我们将使用这些无状态的故事作为交互式学习的起点。

在很多情况下，简单的训练“对话”只是一个转变和回应：“你好”总是会有一个问候，“再见！”总会遇到一个签字，而对“谢谢”的正确回应是几乎总是“不客气”。

以下是故事的摘录。

注意

请注意，下面我们定义了两个故事，显示了这两个故事 `action_show_venue_reviews` 并且 `action_show_concert_reviews` 都是可能的 `compare_reviews` 意图响应，但都没有引用任何上下文。那会晚一点。

```
## greet
* greet
    - utter_greet

## happy
* thankyou
    - utter_youarewelcome
...

## compare_reviews_venues
* compare_reviews
    - action_show_venue_reviews

## compare_reviews_concerts
* compare_reviews
    - action_show_concert_reviews
```

互动学习

运行脚本 `train_online.py`。首先通过将我们提供的故事与更长时间的对话相结合来创建无状态策略，然后对该数据集进行训练。

然后它运行机器人，以便您可以提供反馈来训练它（这是学习*变得互动的地方*）：

快乐的路径

注意

我们在这里没有连接 NLU 工具，所以当你输入消息给机器人时，你必须输入从 a 开始的意图`/`（参见固定意图和实体输入）。如果你想使用 Rasa NLU / wit.ai / Lex，你可以在 `Interpreter` 类中交换 `run.py` 和 `train_online.py`。
我们现在通过直接输入意图开始与机器人交谈。例如，如果我们键入`/greet`，我们会得到以下提示：

```
/greet
------
Chat history:
```

```
    bot did:    None
    bot did:        action_listen
    user said:      /greet

            whose intent is:        greet

we currently have slots: concerts: None, venues: None
------
The bot wants to [utter_greet] due to the intent. Is this correct?

    1.      Yes
    2.      No, intent is right but the action is wrong
    3.      The intent is wrong
    0.      Export current conversations as stories and quit
```

这给你所有的信息，你应该有必要决定什么机器人*应该*做的。在这种情况下，机器人选择了正确的动作（'utter_greet'），所以我们输入 `1` 并按回车。然后我们 `1` 再次输入，因为'action_listen'是问候后的正确动作。我们继续这个循环，直到机器人选择错误的动作。

提供错误反馈

如果你问`/search_concerts`，机器人应该建议 `action_search_concerts`，然后 `action_listen`。现在让我们问一下`/compare_reviews`。机器人碰巧从我们在故事中写到的两种可能性中选择了错误的一种：

```
/compare_reviews
------
Chat history:

    bot did:        action_search_concerts
    bot did:        action_listen
    user said:      /compare_reviews

              whose intent is:     compare_reviews

we currently have slots: concerts: [{'artist': 'Foo Fighters', 'reviews': 4.5}, {'artist':
'Katy Perry', 'reviews': 5.0}], venues: None
------
The bot wants to [action_show_venue_reviews] due to the intent. Is this correct?

    1.      Yes
    2.      No, intent is right but the action is wrong
```

```
3.      The intent is wrong
0.      Export current conversations as stories and quit
```

现在我们输入 `2`，因为它选择了错误的操作，我们得到一个新的提示，询问正确的提示。这也显示了模型分配给每个动作的概率。

在这种情况下，机器人应该 `action_show_concert_reviews`（而不是场地评论！），所以我们键入 `8` 并按回车。

注意

策略模式将会*即时*更新，这样再犯同样的错误的可能性就会降低。您还可以导出所有与机器人的对话，以便将其添加为将来的训练故事。
现在，只要我们想创建更长的对话，我们就可以继续与机器人通话。在任何时候，您都可以键入 `0`，机器人会将当前对话写入文件并退出对话。确保将倾销的故事与您的原始训练数据结合进行下一次训练。

注意

如果你没有足够的训练数据运行机器人，它可能会对 `action_listen` 你的输入产生最可能的反应，因此什么也不做。如果您继续输入内容并得不到答案，请前往互动式训练，并检查是否 `action_listen` 被选为答复。纠正机器人的行为，添加其他故事并运行，`train.py` 然后再次运行机器人。

动机：为什么选择互动学习？

聊天机器人训练有一些复杂性，使其比大多数机器学习问题更棘手。

首先是有几种达到相同目标的方式，而且它们都可以相当好。因此，肯定地说，给定 X，你应该做 Y，如果你不完全 Y，那么你错了。这基本上是你在完全监督学习案例中所做的。我们希望机器人能够学习它可以通过多种不同的方式获得成功的状态。

其次，用户的话语会受到机器人行为的强烈影响。这意味着对预先收集的数据进行训练的网络将遭受暴露偏差。这是一个系统被训练来进行预测的时候，但却从未被赋予训练自己预测的能力，而是每次都被赋予基本事实。在试图预测未来多个步骤的序列时，这已被证明存在问题。

此外，从实际角度来看，Rasa Core 开发人员应该能够通过“ 绿野仙踪”方法进行训练。这意味着如果你想要一个机器人完成某项任务，你可以简单地假装成一个机器人，最后它会学会如何回应。这是学习如何使对话自然流畅的好方法。

没有 Python 的 Rasa Core

注意

在本教程中，我们将构建一个使用 Rasa Core 作为 HTTP 服务器的演示应用程序。尽管您不需要编写任何 Python 代码，但您仍然需要了解域，故事和操作等基本概念。

GitHub 上的示例代码

目标

我们将创建一个简单的机器人，它不要求您使用 python 编写自定义操作代码，而是使用任何其他语言。为了实现这一点，该过程与处理消息有点不同 - 因为 Rasa Core 无法在内部执行操作，但需要等待您执行它们。

我们首先创建一个项目文件夹：

```
mkdir remotebot &&  cd remotebot
```

在本教程之后，文件夹结构应如下所示：

```
remotebot/
├── data/
│   ├── stories.md             # dialogue training data
│   └── concert_messages.md     # nlu training data
├── concert_domain_remote.yml   # dialogue configuration
└── nlu_model_config.json       # nlu configuration
```

创建机器人的第一步与其他 Rasa Core 机器人非常相似。但是，无论如何，我们再来看看它们中的每一个 - 最后我们会到达 HTTP 接口！

1.定义一个域

该域与其他示例类似，但不包含完整的模板。这是因为您需要处理自己的输入和输出连接（例如，发送和接收来自 Facebook 的消息）。

以下是我们 remotebot 的示例域 `concert_domain_remote.yml`：

```
slots:
  concerts:
    type: list
  venues:
    type: list

intents:
  - greet
  - thankyou
  - goodbye
  - search_concerts
  - search_venues
  - compare_reviews

entities:
  - name

action_factory: remote

actions:
  - default
  - greet
  - goodbye
```

```
24  - youarewelcome
25  - search_concerts
26  - search_venues
27  - show_concert_reviews
28  - show_venue_reviews
```

一个重要的区别是。这告诉 Rasa 它不应该自行运行这些操作。

`action_factory: remote`

注意

你可以**选择你喜欢的任何动作名称**。Rasa Core 不会（也不能）检查其有效性。Rasa 将使用这些名称通知您需要执行的操作。所以你需要能够知道如何做一个行动名称。

2.定义一个解释器

我们将使用 Rasa NLU 作为解释器，所以让我们在下面创建一些意图示例 `data/concert_messages.md`:

```
1 ## intent:goodbye
 2 - bye
 3 - goodbye
 4 - good bye
 5 - stop
 6 - end
 7 - farewell
 8 - Bye bye
 9 - have a good one
10
11 ## intent:thankyou
12 - thanks
13 - thank you
14 - thank you very much
15 - nice
16
17 ## intent:search_concerts
18 - I am looking for a concert
19 - where can I find a concert
20 - Whats the next event
21
22 ## intent:search_venues
```

```
- what is a good venue
- where is the best venue
- show me a nice venue
- how about a venue

## intent:compare_reviews
- how do these compare
- what are the reviews
- what do people say about them
- which one is better

## intent:greet
- hey
- howdy
- hey there
- hello
- hi
- good morning
- good evening
- dear sir
```

此外，我们需要 `nlu_model_config.json` NLU 模型的配置文件：

```
{
  "pipeline": "spacy_sklearn",
  "path" : "./models",
  "project": "nlu",
  "data" : "./data/concert_messages.md"
}
```

我们现在可以使用我们的示例来训练 NLU 模型（确保首先 安装 Rasa NLU 以及 spaCy）。

让我们跑吧

```
python -m rasa_nlu.train -c nlu_model_config.json --fixed_model_name current
```

训练我们的 NLU 模型。`models/nlu/current` 应创建一个包含 NLU 模型的新目录。

注意

要收集关于上述配置的更多见解，并且 Rasa NLU 功能将转入 Rasa NLU 文档。

3.定义故事

我们还需要添加几个故事来定义流程（这样我们才能最终获得在远程模式下运行的有趣部分）。让我们把它们放到 `data/stories.md`：

```
## greet
* greet
    - greet

## happy
* thankyou
    - youarewelcome

## goodbye
* goodbye
    - goodbye

## venue_search
* search_venues
    - search_venues
    - slot{"venues": [{"name": "Big Arena", "reviews": 4.5}]}

## concert_search
* search_concerts
    - search_concerts
    - slot{"concerts": [{"artist": "Foo Fighters", "reviews": 4.5}]}

## compare_reviews_venues
* compare_reviews
    - show_venue_reviews

## compare_reviews_concerts
* compare_reviews
    - show_concert_reviews
```

要训　　练对话模型，请运行：

```
python -m rasa_core.train -s data/stories.md -d concert_domain_remote.yml -o models/dialogue
```

这将训练模型并存储 `models/dialogue`。

4.运行服务器

现在我们可以使用我们训练好的对话模型和之前创建的 NLU 模型来运行我们的服务器模式下的机器人：

```
python -m rasa_core.server -d models/dialogue -u models/nlu/current -o out.log
```

我们终于得到它了！僵尸程序正在运行并等待您的 HTTP 请求。

5.使用服务器

所有通信都将通过 HTTP 接口进行。您需要发送一个请求到该接口来开始对话以及您在结束时运行的操作。

包括可用于运行服务器的示例的详细说明可在开始对话中找到。

域，插槽和操作

域

它 `Domain` 定义了你的机器人运行的领域。它明确指出：

- 其中 `intents` 您所期待回应
- 这 `slots` 要跟踪
- 这 `actions` 你的机器人可以采取

例如，`DefaultDomain` 具有以下 yaml 定义：

```
# all hashtags are comments :)
intents:
 - greet
 - default
 - goodbye

entities:
 - name

slots:
  name:
    type: text

templates:
  utter_greet:
    - "hey there {name}!"    # variable will be filled by slot with the same name or by
custom code
```

```
  utter_goodbye:
    - "goodbye :("
    - "bye bye"              # multiple templates will allow the bot to randomly pick from
them
  utter_default:
    - "default message"

actions:
  - utter_default
  - utter_greet
  - utter_goodbye
```

这是什么意思？

`intent`是一个类似于`"greet"`或的字符串`"restaurant_search"`。它描述了你的用户*可能想说的话*。例如，“向我展示墨西哥餐馆”和“我想吃午饭”都可以被描述为一种`restaurant_search`意图。

`slots`是你想在谈话中跟踪的事情。例如，在上面的消息中，您希望将“墨西哥”作为美食类型进行存储。跟踪器有一个像`tracker.get_slot("cuisine")`返回的属性`"Mexican"`

`actions`是你的机器人实际上可以做的事情。通过调用`action.run()`方法调用它们。例如，一个`action`罐子：

- 回应用户
- 进行外部 API 调用
- 查询数据库

注意

有关完整模板格式的更多信息（例如使用像`{name}`或按钮这样的变量）可以查看话语模板。

定义自定义操作

最简单的是`UtterActions`，它只是向用户发送消息。您可以通过将条目添加到以话语命名的操作列表来定义它们。例如，如果应该有一个动作说出发出的模板`utter_greet`，则需要将其添加`utter_greet`到定义的动作列表中。在上面的示例

yaml 中，您可以看到所有三个定义的操作都是以完全模板命名的，因此只需向用户回复一条消息。

更复杂的行动呢？ 要继续使用餐厅示例，如果用户说“让我看一家墨西哥餐馆”，那么您的机器人将执行此操作 `ActionCheckRestaurants`，可能如下所示：

```
from rasa_core.actions import Action
from rasa_core.events import SlotSet

class ActionCheckRestaurants(Action):
   def name(self):
      # type: () -> Text
      return "action_check_restaurants"

   def run(self, dispatcher, tracker, domain):
      # type: (Dispatcher, DialogueStateTracker, Domain) -> List[Event]

      cuisine = tracker.get_slot('cuisine')
      q = "select * from restaurants where cuisine='{0}' limit 1".format(cuisine)
      result = db.query(q)

      return [SlotSet("matches", result if result is not None else [])]
```

请注意，操作**不会直接改变跟踪器**。相反，可以返回 `events` 由跟踪器记录并用于修改其自身状态的操作。

把它放在一起

让我们将这一个新动作添加到自定义域（假设我们将该动作存储在名为的模块中 `restaurant.actions`）：

```
actions:
  - utter_default
  - utter_greet
  - utter_goodbye
  - restaurant.actions.ActionCheckRestaurants   # custom action
```

我们只在这里显示已更改的动作列表，您还需要包含原始域中的其他部分！这一点只是为了展示这些部分如何组合在一起。正如您所看到的，在 `actions` 您的域的部分中，您可以列出完全的动作（向用户响应完整的模板）以及使用其模块路径的自定义动作。

例如，你可以运行，检查构建一个简单的机器人。

话语模板

话语模板是机器人将发回给用户的消息。或者通过与话语名称相同的动作（例如，在上面的示例中，使用 *utter_default* 模板和动作）或通过使用自定义代码的动作自动完成。

图像和按钮

在域 yaml 文件中定义的模板也可以包含图像和按钮：

```
templates:
  utter_greet:
  - text: "Hey! How are you?"
    buttons:
    - title: "great"
      payload: "great"
    - title: "super sad"
      payload: "super sad"
  utter_cheer_up:
  - text: "Here is something to cheer you up:"
    image: "https://cdn77.eatliver.com/wp-content/uploads/2017/10/trump-frog.jpg"
```

注意

请记住，如何显示定义的按钮取决于输出通道的实现。例如，cmdline 界面无法显示按钮或图像，但会尝试在命令行中模拟它们。

变量

您也可以使用模板中的**变量**插入对话期间收集的信息。您可以在自定义 Python 代码中执行该操作，也可以使用自动槽填充机制。例如，如果你有这样的模板：

```
templates:
  utter_greet:
  - text: "Hey, {name}. How are you?"
```

Rasa 将自动填充该变量，并在所调用的插槽中找到一个值 `name`。

在自定义代码中，您可以使用以下方法检索模板：

```
class ActionCustom(Action):
   def name(self):
```

```
        return "action_custom"

    def run(self, dispatcher, tracker, domain):
        # send utter default template to user
        dispatcher.utter_template("utter_default")
        # ... other code
        return []
```

如果模板包含用`{my_variable}`您指定的变量，则可以通过将字段作为关键字参数传递给 `utter_template`：

```
dispatcher.utter_template("utter_default", my_variable="my text")
```

变化

如果你想随机改变发送给用户的响应，你可以列出多个响应，机器人将随机选择其中的一个响应，例如：

```
templates:
  utter_greeting:
  - text: "Hey, {name}. How are you?"
  - text: "Hey, {name}. How is your day going?"
```

插槽

大多数插槽影响机器人应运行的下一个动作的预测。对于预测，槽值不是直接使用，而是具有特征。例如，对于一个类型的槽来说 `text`，该值是无关紧要的，因为对于设计而言，唯一重要的是如果文本被设置或不设置。在 `unfeaturized` 任何值的槽中都可以存储，该槽不会影响预测。

插槽类型的选择应该小心。如果一个时隙值应该影响对话流程（例如，用户年龄影响下一个问题），您应该选择一个时间值来影响对话模型。

这些都是预定义的插槽类以及它们对以下内容有用的内容：

text

用于： 用户喜好，你只关心他们是否被指定。

例：

```
slots :
   internal_user_id :
      type  :  unfeaturized
```

描述： `1` 如果设置了任何值，则将插槽的功能设置为。否则，该功能将被设置为 `0`（没有设置值）。

bool

用于： 对或错

例：

```
slots:
   is_authenticated:
      type: bool
```

描述： 检查是否设置了插槽并且是否为真

categorical

用于： 插槽可以取 N 个值中的一个

例：

```
slots:
   risc_level:
      type: categorical
      values:
      - low
      - medium
      - high
```

描述： 创建一个热点编码，描述哪个 `values` 匹配。

float

用于： 连续值

例：

```
slots:
   temperature:
      type: float
      min_value: -100.0
      max_value:  100.0
```

默认值： `max_value=1.0`， `min_value=0.0`

描述： 以下所有值 `min_value` 将被视为 `min_value`，上述值也是如此 `max_value`。因此，如果 `max_value` 设置为 `1`，槽值 `2` 和特征之间没有区别 `3.5`（例如，两个值将以相同的方式影响对话，

并且模型不能学会区分它们）。

list

用于： 值列表

例：

```
slots :
  shopping_items :
    type :  list
```

描述： 此插槽的功能设置为，`1` 如果列表中的值已设置，列表不为空。如果未设置任何值，或者空白列表是设置值，则该功能将为 `0`。将**存储在槽不影响对话列表的长度**。

unfeaturized

用于： 您要存储的数据不应影响对话流程

例：

```
slots :
  internal_user_id :
    type :  unfeaturized
```

描述： 这个插槽不会有任何特征，因此它的价值不会影响对话流程，并且在预测机器人应该运行的下一个动作时被忽略。

data

用于： 创建自己的插槽的基类

例：

告

这种类型不应该直接使用，而应该被分类。

描述： 用户必须对此进行子类化并定义 `as_feature` 包含任何自定义逻辑的方法。

故事 - 训练数据

对话系统的训练数据样本被称为**故事**。这将向您展示如何定义它们以及如何将它们可视化。

格式

以下是 bAbi 数据的一个例子：

```
## story_07715946
```

```
* greet
   - action_ask_howcanhelp
* inform{"location": "rome", "price": "cheap"}
   - action_on_it
   - action_ask_cuisine
* inform{"cuisine": "spanish"}
   - action_ask_numpeople
* inform{"people": "six"}
   - action_ack_dosearch
```

这就是我们所说的**故事**。一个故事从一个名字开始，前面有两个哈希，这是任意的，但可以用于调试。故事的结尾用换行符表示，然后重新开始一个新故事。

```
## story_03248462##
```

您可以使用模块化和简化您的训练数据：`> checkpoints`

```
## first story
* hello
   - action_ask_user_question
> check_asked_question

## user affirms question
> check_asked_question
* affirm
  - action_handle_affirmation

## user denies question
> check_asked_question
* deny
  - action_handle_denial
```

故事的可视化

有时候可以对故事文件中描述的会话路径进行概述。为了简化调试并简化关于 bot 流的讨论，您可以将故事文件的内容可视化。

注意

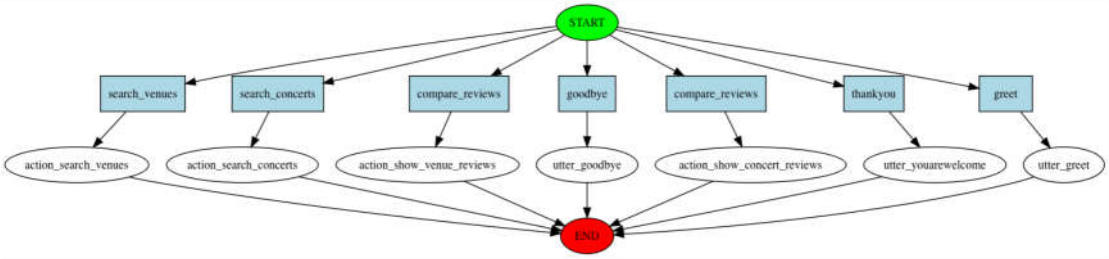
为了这个工作，你需要**安装 graphviz**。这些是在 OSX 上执行该操作的说明，对于其他系统，说明可能略有不同：

```
brew install graphviz
pip install pygraphviz --install-option="--include-path=/usr/include/graphviz" \
  --install-option="--library-path=/usr/lib/graphviz/"
```

只要安装完毕，您可以将这样的故事可视化：

```
cd examples/concertbot/
python -m rasa_core.visualize -d concert_domain.yml -s data/stories.md -o graph.png
```

这将贯穿 `concertbot` 示例中 的故事 `data/stories.md` 并创建存储在输出图像中的图形 `graph.png`。



我们也可以直接从代码运行可视化。对于这个例子，我们可以用下面的代码创建一个 `visualize.py`in `examples/concertbot`：

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

from rasa_core.agent import Agent
from rasa_core.policies.keras_policy import KerasPolicy
from rasa_core.policies.memoization import MemoizationPolicy

if __name__ == '__main__':
    agent = Agent("concert_domain.yml",
                  policies=[MemoizationPolicy(), KerasPolicy()])

    agent.visualize("data/stories.md",
                    output_file="graph.png", max_history=2)
```

这将创建与上面的 python 脚本调用相同的图像。所显示的图形仍然非常简单，但图形可能会变得非常复杂。

如果您想从故事文件中替换故事文件，这些故事文件通常看起来像是 `greet` 真实的信息，例如 `Hello`，您可以传入 Rasa NLU 训练数据实例，以将它们替换为训练数据中的消息。

注意

故事可视化需要加载你的域名。如果你有任何用 python 编写的自定义动作，请确保它们是 python 路径的一部分，并且可以通过可视化脚本使用为域中的动作提供的模块路径加载（例如 `actions.ActionSearchVenues`）。

常见模式

注意

在这里，我们通过一些典型的对话模式并解释如何使用 Rasa Core 来实现它们。Rasa Core 使用 ML 模型来预测行为，并 `slots` 提供这些模型依赖的重要信息来跟踪上下文并处理多回合对话。插槽用于存储与多个轮次相关的信息。例如，在我们的餐厅示例中，我们希望跟踪对话期间的美食和人数等情况。

收集信息以完成请求

为了收集一系列偏好，您可以 `TextSlot` 像在餐厅示例中那样使用：

```
slots :
  cuisine :
```

```
    type :  text
people : type : text ...
```

当 Rasa 看到与其中一个插槽具有相同名称的实体时，该值将自动保存。例如，如果你的 NLU 模块 `people=8` 在句子*“我想要一个8人桌”*的句子中检测到这个实体，它将被保存为一个插槽，

```
>>> tracker.slots
{'people': <TextSlot(people: 8)>}
```

当 Rasa Core 预测下一个要采取的行动时，关于 `TextSlot` 的唯一信息是 **他们是否被定义**。因此，您有足够的信息知道您不必再次询问这些信息，但 `TextSlot` 的*值*对 Rasa Core 预测的操作没有影响。这在下面更详细地解释。

这里描述了全套插槽类型及其行为：插槽。

使用槽值来影响哪些行为被预测

自定义插槽

也许您的餐厅预订系统只能处理最多 6 人的预订，所以上述请求无效。在这种情况下，您希望插槽的*值*影响下一个选定的操作（而不仅仅是是否已指定）。您可以使用自定义插槽类实现　　此目的。

我们在下面定义它的方式，如果人数少于或等于 6，我们会返回`(1,0)`，如果我们返回更多`(0,1)`，并且没有设置`(0,0)`。

Rasa Core 可以使用这些信息来区分不同的情况 - 只要您有一些训练故事可以在适当的响应中进行，例如：

```
# story1
...
* inform{"people": "3"}
- action_book_table
...
# story2
* inform{"people": "9"}
- action_explain_table_limit
```

```
from rasa_core.slots import Slot
```

```
class NumberOfPeopleSlot(Slot):

  def feature_dimensionality(self):
      return 2

  def as_feature(self):
      r = [0.0] * self.feature_dimensionality()
      if self.value:
          if self.value <= 6:
              r[0] = 1.0
          else:
              r[1] = 1.0
      return r
```

如果你想存储类似价格范围的东西，这实际上更简单一些。像价格范围这样的变量通常具有一个 n 值，例如低，中，高。对于这些情况，您可以使用 `categorical` 插槽。

```
slots:
  price_range:
    type: categorical
    values: low, medium, high
```

Rasa 自动表示（featurises）此作为值的一热编码：`(1,0,0)`，`(0,1,0)`，或`(0,0,1)`。

插槽功能

当 Rasa Core 使用故事训练对话模型时，`Slot`将使用条目的存在来帮助确定下一个应该采取的动作。这最适合`CategoricalSlot`插槽类型。

`TextSlot`可以有任何价值，但它只有一个特征。它可以被设置，在这种情况下该特征有一个值`(1)`，或者如果没有设置它将会有一个值`(0)`。

`CategoricalSlot`有许多值，每个值都是一个特征。以下面的例子为例，当`restaurant_availability`插槽被设置时，Rasa Core 将能够确定所讨论的餐厅是否可用，并根据该值选择完全不同的动作来执行。

```
restaurant_availability:
    type: categorical
    values:
    - unknown
    - booked-out
    - waiting-list
```

```
    - available
```

`Slot` 如果其名称和 NLU 模块检测到的实体名称匹配，则 Rasa Core 将设置 A. 如果您将插槽添加到训练故事中，插槽的值将影响故事对话 - 这在下面的示例中进行了解释。插槽也可以根据我们自己的习惯明确设置，`Action` 并根据真实世界的信息影响对话。

```
class ActionMakeBooking(Action):

    def run(self, dispatcher, tracker, domain):
        restaurant_name=tracker.get_slot("restaurant_name")
        location=tracker.get_slot("location")
        num_people=tracker.get_slot("people")
        date=tracker.get_slot("date")
        # this will fetch the availability of the restaurant from your DB or an API
        availability=restaurantService.check_availability(restaurant_name, location,
num_people, date)
        return [SlotSet("restaurant_availability", availability)]
```

假设的代码片段 `Action` 显示了插槽的值 `restaurant_availability` 是通过查询数据库或 API 来确定的。当我们训练对话模型时，餐厅的可用性不是已知的，`Slot` 价值是我们根据外部信息改变对话过程的唯一方法。

从 API 调用中获取的数据也可以存储起来以备后用，而不会改变会话结果，详见在跟踪器中存储 API 响应。

插槽功能示例

注意

为了便于说明，这些示例故事已经过手动构建。虽然这是一种训练模型的有效方法，但首选的方法是使用交互式学习来生成*更*不容易出错的故事。

在第一个故事中，我们将尝试在 2018 年 8 月 21 日晚上在一家餐厅为 5 个人预订。在这种情况下，餐厅已预订完毕，因此我们想向客户道歉并建议类似的餐厅。假设 Rasa Core 模型已经被训练识别一个消息，如*“8 月 21 日为 5 人预订 Murphys Bistro”*

```
# restaurant unavailable
* _make_booking{"people":"5", "date":"2018-08-21T19:30:00+00:00",
"restaurant_id":"145"}
- slot{"restaurant_availability": "booked-out"}
- utter_sorry_unavailable
- action_show_similar
```

这第二个故事详细介绍了餐厅可用时的流程。我们会告诉客户我们已经预订了餐厅，并询问是否需要进一步帮助。

在最后一个例子中，`make_booking`找到了意图，但是 Rasa Core 未能解析日期或未提供日期。在这种情况下，我们需要索取更多信息。

注意：这最后一个故事是使用事实`date`是一个`TextSlot`，因此有一个单独的功能是否设置。

```
# restaurant request without date
* _make_booking{"people":"5", "restaurant_id":"145"}
- slot{"date": null}
- utter_date_required
* _inform{"date":"2018-08-22T19:30:00+00:00"}
- action_make_booking
- utter_restaurant_booked
- utter_anything_more
* _bye
- utter_thank_you
```

将 API 响应存储在跟踪器中

从 API 调用的结果可以被存储在一个`Slot`作为上面所解释的。在这种情况下，数据被存储在一个`Slot`具有特色的图像中，影响对话的流程。

`unfeaturized`可以使用类型的槽来存储来自数据库查询或 API 调用的结果，以便它不会影响对话的过程。`unfeaturized`在域文件中定义的一个实例槽：

```
slots :
    api_result :
        type :  unfeaturized
```

您可以在自定义中设置此值`Action`：

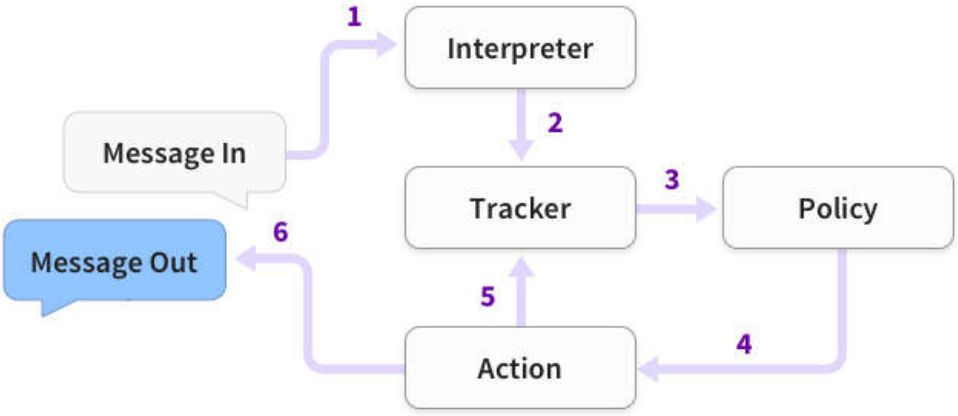
```
from rasa_core.actions import Action
from rasa_core.events import SlotSet
import requests

class ApiAction(Action):
    def name(self):
        return "api_action"

    def run(self, tracker, dispatcher):
```

```
        data = requests.get(url).json
        return [SlotSet("api_result", data)]
```

当您在 Redis 或其他数据存储区中保留跟踪器时，此功能尤其有用。您可以单独缓存 API 或数据库响应，但将它们存储在跟踪器中意味着它们将随对话状态的其余部分自动保持，并且在系统需要重新启动时将与状态的其余部分一起恢复。

探究 - 如何融合在一起

此图显示了 Rasa Core 应用程序如何响应消息的基本步骤：



步骤是：

1. 该消息被接收并传递给一个 `Interpreter`，该消息将其转换成包含原始文本，意图以及找到的任何实体的字典。
2. 这 `Tracker` 是跟踪对话状态的对象。它收到新消息进入的信息。
3. 该策略会收到跟踪器的当前状态
4. 策略选择接下来要采取的行动。
5. 所选择的操作由跟踪器记录
6. 一个响应被发送给用户

注意

消息可以是由人类输入的文本，但同样也可以是按钮负载，或者是已经被解释的消息（就像你从 Amazon Echo 获得的消息一样）。有关详细信息，请参阅解释器部分。

上述步骤由 `MessageProcessor` 接收消息并在处理消息时进行。

随意查看构建一个简单的机器人和 互动学习，看看一个机器人的行动！

HTTP 服务器

注意

在使用服务器之前，您需要定义一个域，创建训练数据并训练一个模型。然后，您可以使用训练好的模型进行远程代码执行！有关介绍，请参阅构建简单的 Bot。

警告

HTTP API 仍然是实验性的，我们希望您的反馈（例如通过 Gitter）。

HTTP api 的存在使非 python 项目可以轻松使用 Rasa Core。

概观

总体思路是在代码中运行这些动作（任意语言），而不是 python。为此，Rasa Core 将启动一个 Web 服务器，您需要将用户消息传递给该服务器。另一方面，Rasa Core 会告诉你需要运行哪些操作。在运行这些操作之后，您需要通知框架执行它们，并告诉模型关于该用户的内部对话状态的任何更新。所有这些交互都使用 HTTP REST 接口完成。

要激活远程模式，请包括

```
action_factory: remote
```

在你的 `domain.yml`（你可以找到一个例子 `examples/remote/concert_domain_remote.yml`）。

注意

如果作为 HTTP 服务器启动，Rasa Core 将不会为您处理输出或输入通道。这意味着你需要从输入频道（例如 facebook messenger）获取消息，并将消息发送给用户。

因此，你也不需要在你的域 yaml 中定义任何话语。只需列出你需要的所有动作。

运行服务器

您可以运行一个简单的 http 服务器来处理使用您的模型的请求

```
$ python -m rasa_core.server -d examples/babi/models/policy/current -u
examples/babi/models/nlu/current_py2 -o out.log
```

不同的参数是：

- `-d`，这是通往 Rasa Core 模型的路径。
- `-u`，这是通往 Rasa NLU 模型的路径。

- `-o`，这是日志文件的路径。

开始对话

你需要做一个 `POST` 到`/conversation/<sender_id>/parse`终点。 `<sender_id>`是对话 ID（例如，`default`如果你只有一个用户，或 Facebook 用户 ID 或任何其他标识符）。

```
$ curl -XPOST localhost:5005/conversations/default/parse -d '{"query":"hello there"}'
```

服务器将回应你应该采取的下一个动作：

```
{
  "next_action": "utter_ask_howcanhelp",
  "tracker": {
    "slots": {
      "info": null,
      "cuisine": null,
      "people": null,
      "matches": null,
      "price": null,
      "location": null
    },
    "sender_id": "default",
    "latest_message": {
      ...
    }
  }
}
```

您现在需要 `utter_ask_howcanhelp`在您的最后执行操作。这可能包括发送消息到输出频道（例如回到 facebook）。

在完成上述操作之后，您需要通知 Rasa Core：

```
$ curl -XPOST http://localhost:5005/conversations/default/continue -d \
   '{"executed_action": "utter_ask_howcanhelp", "events": []}'
```

这里的 API 应该回应：

```
{
  "next_action":"action_listen",
  "tracker": {
    "slots": {
      "info": null,
      "cuisine": null,
```

```
      "people": null,
      "matches": null,
      "price": null,
      "location": null
    },
    "sender_id": "default",
    "latest_message": {
      ...
    }
  }
}
```

该响应告诉您等待下一个用户消息。在收到包含 `action_listen` 下一个操作的响应后，您不应该调用 continue 端点。相反，请等待下一条用户消息，`/conversations/default/parse` 然后再次调用 ，然后再次调用，`/conversations/default/continue` 直至 `action_listen` 再次出现。

事件

事件允许您修改对话的内部状态。这些信息将被用来预测下一个动作。例如，您可以设置插槽（以存储有关用户的信息）或重新启动对话。

作为查询的一部分，您可以返回多个事件，例如：

```
$ curl -XPOST http://localhost:5005/conversations/default/continue -d \
    '{"executed_action": "search_restaurants", "events": [{"event": "slot", "name":
"cuisine", "value": "mexican"}, {"event": "slot", "name": "people", "value": 5}]}'
```

以下是您可以 `events` 在呼叫中追加到阵列的所有可用事件的列表 `/conversation/<sender_id>/continue`。

设置一个插槽

名称： `slot`

例子： `"events": [{"event": "slot", "name": "cuisine", "value": "mexican"}]`

描述： 将插槽的值设置为传递的值。根据插槽类型，您设置的值应该合理。

重新开始

名称： `restart`

例子： `"events": [{"event": "restart"}]`

描述： 重新开始对话并重置所有插槽和过去的操作。

重置插槽

名称： `reset_slots`

例子： `"events": [{"event": "reset_slots"}]`

描述： 将所有插槽重置为其初始值。

端点

POST /conversations/（*str：sender_id*）/parse

通知用户发布新消息的对话引擎。你必须 `POST` 以这种格式存储数据，你可以这样做`'{"query":"<your text to parse>"}'`

示例请求：

```
curl -XPOST localhost:5005/conversations/default/parse -d \
  '{"query":"hello there"}' | python -mjson.tool
```

响应示例：

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
    "next_action": "utter_ask_howcanhelp",
    "tracker": {
        "latest_message": {
            ...
        },
        "sender_id": "default",
        "slots": {
            "cuisine": null,
            "info": null,
            "location": null,
            "matches": null,
            "people": null,
            "price": null
        }
```

```
    }
}
```

状态码：

- 200 OK - 没有错误

POST /conversations/（str：sender_id）/continue

继续使用 id 为 *user_id* 的对话的预测循环。应该被调用，直到端点返回 `action_listen` 作为下一个动作。在对这个端点的调用之间，你的代码应该执行下面提到的动作。如果您收到 `action_listen` 下一个动作，则应等待下一个用户输入。

示例请求：

```
curl -XPOST http://localhost:5005/conversations/default/continue -d \
  '{"executed_action": "utter_ask_howcanhelp", "events": []}' | python
-mjson.tool
```

响应示例：

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
    "next_action": "utter_ask_cuisine",
    "tracker": {
        "latest_message": {
            ...
        },
        "sender_id": "default",
        "slots": {
            "cuisine": null,
            "info": null,
            "location": null,
            "matches": null,
            "people": null,
            "price": null
        }
    }
}
```

Status Codes:

- 200 OK – no error

GET /conversations/（str：sender_id）/tracker

使用以下方式检索对话的当前跟踪器状态 `sender_id`。这包括设置的插槽以及最新消息和所有以前的事件。

示例请求：

```
curl http://localhost:5005/conversations/default/tracker | python -mjson.tool
```

响应示例：

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
    "events": [
        {
            "event": "action",
            "name": "action_listen"
        },
        {
            "event": "user",
            "parse_data": {
                "entities": [],
                "intent": {
                    "confidence": 0.7561643619088745,
                    "name": "affirm"
                },
                "intent_ranking": [
                    ...
                ],
                "text": "hello there"
            },
            "text": "hello there"
        }
    ],
    "latest_message": {
        "entities": [],
        "intent": {
            "confidence": 0.7561643619088745,
            "name": "affirm"
        },
```

```
            "intent_ranking": [
                ...
            ],
            "text": "hello there"
        },
        "paused": false,
        "sender_id": "default",
        "slots": {
            "cuisine": null,
            "info": null,
            "location": null,
            "matches": null,
            "people": null,
            "price": null
        }
    }
```

Status Codes:

- 200 OK – no error

PUT /conversations/（str：sender_id）/tracker

使用事件替换跟踪器状态。任何现有的跟踪器 `sender_id` 都将被丢弃。将创建一个新的跟踪器，并将传递的事件应用于创建一个新状态。

传递事件的格式与 `/continue` 端点的格式相同。

示例请求：

```
curl -XPUT http://localhost:5005/conversations/default/tracker -d \
  '[{"event": "slot", "name": "cuisine", "value": "mexican"},{"event":
"action", "name": "action_listen"}]' | python -mjson.tool
```

响应示例：

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
    "events": [
        {
            "event": "slot",
            "name": "cuisine",
            "value": "mexican"
        },
```

```
            {
                "event": "action",
                "name": "action_listen"
            }
        ],
        "latest_message": {
            "entities": [],
            "intent": {},
            "text": null
        },
        "paused": false,
        "sender_id": "default",
        "slots": {
            "cuisine": "mexican",
            "info": null,
            "location": null,
            "matches": null,
            "people": null,
            "price": null
        }
    }
```

Status Codes:

- 200 OK – no error

POST /conversations/（*str：sender_id*）/tracker/events

将事件跟踪器状态追加到事件。任何现有的事件将被保留并且新事件将被追加，更新现有状态。

传递事件的格式与`/continue`端点的格式相同。

示例请求：

```
curl -XPOST http://localhost:5005/conversations/default/tracker/events -d \
  '[{"event": "slot", "name": "cuisine", "value": "mexican"},{"event":
"action", "name": "action_listen"}]' | python -mjson.tool
```

响应示例：

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
```

```
    "events": null,
    "latest_message": {
        "entities": [],
        "intent": {
            "confidence": 0.7561643619088745,
            "name": "affirm"
        },
        "intent_ranking": [
            ...
        ],
        "text": "hello there"
    },
    "paused": false,
    "sender_id": "default",
    "slots": {
        "cuisine": "mexican",
        "info": null,
        "location": null,
        "matches": null,
        "people": null,
        "price": null
    }
}
```

Status Codes:

- 200 OK – no error

GET /version

当前正在运行的 Rasa Core 版本。

示例请求：

```
curl http://localhost:5005/version | python -mjson.tool
```

响应示例：

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
    "version" : "0.7.0"
}
```

Status Codes:

- 200 OK – no error

安全考虑

我们建议不要将 Rasa Core 服务器暴露给外部世界，而是通过专用连接（例如 Docker 容器之间）从后端连接到它。

不过，还有内置令牌认证。如果您在启动服务器时指定了令牌，则需要在每个请求中传递该令牌：

```
$ python -m rasa_core.server --auth_token thisismysecret -d
examples/babi/models/policy/current -u examples/babi/models/nlu/current_py2 -o out.log
```

在我们的例子中 `thisismysecret`，您的请求应该传递令牌作为参数：

```
$ curl -XPOST localhost:5005/conversations/default/parse?token=thisismysecret -d
'{"query":"hello there"}'
```

连接到消息和语音平台

以下是如何将对话式 AI 与外部世界连接起来。

输入通道在 `rasa_core.channels` 模块中定义。目前，有一个命令行的实现，以及连接到 Facebook，松弛和电报。

Facebook Messenger 安装程序

使用运行脚本

如果您想使用运行脚本连接到 Facebook，例如使用

```
python -m rasa_core.run -d models/dialogue -u models/nlu/current \
    --port 5002 --connector facebook --credentials fb_credentials.yml
```

您需要提供 `fb_credentials.yml` 以下内容：

```
verify: "rasa-bot"
secret: "3e34709d01ea89032asdebfe5a74518"
page-access-token: "EAAbHPa7H9rEBAAuFk4Q3gPKbDedQnx4djJJ1JmQ7CAqO4iJKrQcNT0wtD"
```

直接使用 python

一个 `FacebookInput` 实例提供用于创建 Web 服务器的烧瓶蓝图。这可以让您从 Web 服务器创建逻辑中分离确切的端点和实现。

创建 Messenger 兼容 Web 服务器的代码如下所示：

```
from rasa_core.channels import HttpInputChannel
from rasa_core.channels.facebook import FacebookInput
from rasa_core.agent import Agent
from rasa_core.interpreter import RegexInterpreter

# load your trained agent
agent = Agent.load("dialogue", interpreter=RegexInterpreter())

input_channel = FacebookInput(
    fb_verify="YOUR_FB_VERIFY",  # you need tell facebook this token, to confirm your
URL
    fb_secret="YOUR_FB_SECRET",  # your app secret
    fb_access_token="YOUR_FB_PAGE_ACCESS_TOKEN"   # token for the page you subscribed
to
)agent.handle_channel(HttpInputChannel(5004, "/app", input_channel))
```

参数 `HttpInputChannel` 是端口，URL 前缀和输入通道。接收 Facebook 信使消息的默认端点是 `/webhook`，所以上面的示例将监听消息 `/app/webhook`。这是您应该在 Facebook 开发人员门户中添加的网址。

注意

如何获得 FB 凭证：您需要设置 Facebook 应用程序和页面。

1. 要创建应用程序，请转到：https://developers.facebook.com/并点击*“添加新应用程序”*。
2. 进入应用程序的仪表板并在*产品*下，点击*添加产品*并*添加 Messenger*。在 Messenger 的设置下，向下滚动到*令牌生成*，然后单击该链接为您的应用程序创建一个新页面。
3. 使用收集的 `verify`，`secret` 和你的机器人连接到 Facebook。`access token`

有关更详细的步骤，请访问 信使文档。

闲置设置

使用运行脚本

如果您想使用运行脚本连接到松弛的输入通道，例如使用

```
python -m rasa_core.run -d models/dialogue -u models/nlu/current \
    --port 5002 --connector slack --credentials slack_credentials.yml
```

您需要提供 `slack_credentials.yml` 以下内容：

```
slack_token : “xoxb-286425452756-safjasdf7sl38KLls”
slack_channel : “@my_channel”
```

直接使用 python

一个 `SlackInput` 实例提供用于创建 Web 服务器的烧瓶蓝图。这可以让您从 Web 服务器创建逻辑中分离确切的端点和实现。

创建 Messenger 兼容 Web 服务器的代码如下所示：

```
from rasa_core.channels import HttpInputChannel
from rasa_core.channels.slack import SlackInput
from rasa_core.agent import Agent
from rasa_core.interpreter import RegexInterpreter

# load your trained agent
agent = Agent.load("dialogue", interpreter=RegexInterpreter())

input_channel = SlackInput(
   slack_token="YOUR_SLACK_TOKEN",  # this is the `bot_user_o_auth_access_token`
   slack_channel="YOUR_SLACK_CHANNEL"  # the name of your channel to which the bot
posts
)
agent.handle_channel(HttpInputChannel(5004, "/app", input_channel))
```

参数 `HttpInputChannel` 是端口，URL 前缀和输入通道。接收 Facebook 信使消息的默认端点是 `/webhook`，所以上面的示例将监听消息 `/app/webhook`。这是您应该在 Facebook 开发人员门户中添加的网址。

注意

如何获得 Slack 凭证：您需要设置一个 Slack 应用程序。

1. 要创建应用程序，请转到：https://api.slack.com/apps 并点击“*创建新应用程序*”。
2. 激活以下功能：交互式组件，事件订阅，机器人的用户，权限（基本功能，你应该订阅 `message.channel`， `message.groups`， `message.im` 和 `message.mpim` 事件）
3. 这 `slack_channel` 是你的机器人发布的目标。这可以是频道，应用程序或个人
4. 使用“OAuth＆Permissions”选项卡中的条目作为您的 `Bot User OAuth Access Tokenslack_token`

有关更详细的步骤，请访问 slack api 文档。

电报设置

使用运行脚本

如果您想使用运行脚本连接到松弛的输入通道，例如使用

您需要提供 `telegram_credentials.yml` 以下内容：

```
access_token: "490161424:AAGlRxinBRtKGb21_rlOEMtDFZMXBl6EC0o"
verify: "your_bot"
webhook_url: "your_url.com/webhook"
```

直接使用 python

一个 `TelegramInput` 实例提供用于创建 Web 服务器的烧瓶蓝图。这使您可以从 Web 服务器创建逻辑中分离确切的端点和实现。

创建 Messenger 兼容 Web 服务器的代码如下所示：

```
from rasa_core.channels import HttpInputChannel
from rasa_core.channels.telegram import TelegramInput
from rasa_core.agent import Agent
from rasa_core.interpreter import RegexInterpreter

# load your trained agent
agent = Agent.load("dialogue", interpreter=RegexInterpreter())

input_channel = TelegramInput(
  access_token="YOUR_ACCESS_TOKEN", # you get this when setting up a bot
  verify="YOUR_TELEGRAM_BOT", # this is your bots username
  webhook_url="YOUR_WEBHOOK_URL" # the url your bot should listen for messages
)
```

```
14
15 agent.handle_channel(HttpInputChannel(5004, "/app", input_channel))
```

参数 `HttpInputChannel` 是端口，URL 前缀和输入通道。接收消息的默认端点是 `/webhook`，所以上面的例子会监听消息 `/app/webhook`。这是你应该使用的 URL `webhook_url`，例如 `webhook_url=myurl.com/app/webhook`。为了让 bot 监听该 URL 处的消息，请 `myurl.com/app/set_webhook` 首先设置 webhook。

注意

如何获取电报凭证：您需要设置电报机器人。

1.要创建 bot，请转到：https://web.telegram.org/#/im ？p = @ BotFather，输入 */ newbot* 并按照说明操作。2.最后你应该得到你 `access_token` 的名字和你设定的用户名 `verify`。3.如果您想在群组设置中使用您的机器人，建议您通过输入*/ setprivacy* 打开群组隐私模式。然后，机器人只会在消息以*/ bot* 开始时才会收听
有关 Telegram HTTP API 的更多信息，请访问 https://core.telegram.org/bots/api

计划提醒

注意

这是一个仅限 python 的功能。目前，如果您将框架作为 HTTP 服务器运行，则无法使用调度功能。

您的其中一位用户在完成任务后中途停止了与您的机器人通信。现在你想提醒他们回到你身边。别担心！Rasa 已经替你考虑到了。

要安排稍后执行的操作，将会有一个特殊事件调用 `Reminder`。让我们举一个更具体的例子：我们需要一个确认书来预订一张餐厅的餐桌 - 如果没有它，预订就不会进行。

```
## book a restaurant
* start_booking
    - action_suggest_restaurant
* select_restaurant{"name": "Papi's Pizza Place"}
    - action_confirm_booking
```

这个故事中的最后一个动作是 `action_confirm_booking` 要求用户进行确认，通常用户会直接发送确认信息 - 但有时他们可能会忘记这么做。

我们需要做两件事：

1. 在将来的某个特定时间安排提醒
2. 定义当提醒被触发时会发生什么

安排提醒

提醒将在执行操作后作为事件返回时进行安排。以下是我们的一个示例实现 `action_confirm_booking`:

```
from rasa_core.actions import Action
from rasa_core.events import Reminder
import datetime
from datetime import timedelta

class ActionConfirmBooking(Action):
    def name(self):
```

```
        return "action_confirm_booking"

    def run(self, dispatcher, tracker, domain):
        dispatcher.utter_message("Do you want to confirm your booking at Papi's pizza?")
        return [Reminder("action_booking_reminder", datetime.now() +
timedelta(hours=5))]
```

此操作会在 5 个小时内安排提醒。提醒将触发该操作 `action_booking_reminder`。

当提醒被触发时会发生什么

首先，名字是提醒一部分的动作将被执行。我们的示例的实现可能如下所示：

```
from rasa_core.actions import Action
from rasa_core.events import Reminder


class ActionBookingReminder(Action):
    def name(self):
        return "action_booking_reminder"

    def run(self, dispatcher, tracker, domain):
        dispatcher.utter_message("You have an unconfirmed booking at Papi's pizza, would
you like to confirm it?")
        return []
```

默认情况下，如果用户在计划的提醒时间之前向机器人发送*任何*消息，提醒将被取消。如果你不想发生这种情况，你需要 `kill_on_user_message` 在创建提醒时设置标志：

```
Reminder("action_booking_reminder", datetime.now() + timedelta(hours=5), kill_on_us
er_message=False)
```

在动作被触发后，动作执行继续，就好像用户发送了一个空信息。所以在我们的故事中，我们需要定义提醒执行后会发生什么：

```
## book a restaurant
* start_booking
    - action_suggest_restaurant
* select_restaurant{"name": "Papi's Pizza Place"}
    - action_confirm_booking

## reminder_confirm
    - action_booking_reminder
```

```
* agree
    - action_book_restaurant

## reminder_cancel
    - action_booking_reminder
* deny
    - action_cancel_booking
```

我们添加了两个故事：一个是用户同意我们在提醒中发送的消息，另一个是他们决定取消预订的消息。

警告

这是**非常重要的**指定提醒被触发后会发生什么。否则，在运行提醒动作之后，机器人将不知道该做什么，它会运行一个看似随机的动作。因此，请务必在您的训练数据中添加一个以提醒操作开始的故事。

代理人

代理允许您训练模型，加载并使用它。这是一个使用简单 API 访问大多数 Rasa Core 功能的门面。

注意

并非所有功能都通过代理上的方法公开。有时您需要自行编排不同的组件（域，策略，解释器和跟踪器商店）以对其进行自定义。
开始了：

***class* rasa_core.agent.Agent（*域，策略= None，featurizer = None，interpreter = None，tracker_store = None*）[source]**

基地： `object`

公共接口用于常见的事情。

这包括例如训练助理，或与助理处理消息。

continue_message_handling（*sender_id，executed_action，events*）[source]

***classmethod*create_tracker_store（*store，domain*）[source]**

handle_channel（*input_channel，message_preprocessor = None*）[source]

处理来自频道的消息。

handle_message（*text_message，message_preprocessor = None，output_channel = None，sender_id = u'default'*）[source]

处理一条消息。

如果传递一个消息预处理器，消息将首先传递给该函数，然后将返回值用作对话引擎的输入。

该函数的返回值取决于 *output_channel*。如果输出通道未设置，则设置为 *无*，或设置为 *CollectingOutputChannel*，此功能将返回机器人想要响应的消息。

例：

```
>>> from rasa_core.agent import Agent
>>> agent = Agent.load("examples/restaurantbot/models/dialogue",
... interpreter="examples/restaurantbot/models/nlu/current")
>>> agent.handle_message("hello")
[u'how can I help you?']
```

classmethod **load**（*path*，*interpreter = None*，*tracker_store = None*，*action_factory = None* ）[source]

persist（*model_path* ）[source]

将此代理保留到目录中供以后加载和使用。

start_message_handling（*text_message*，*sender_id = u'default'* ）[source]

toggle_memoization（*activate*）[source]

打开和关闭记忆功能。

如果集体中存在记忆策略，则会切换对该策略的预测。当设置为 *false* 时，策略集合中存在的 Memoization 策略将不作任何预测。因此，来自合奏的预测结果总是需要来自不同的策略（例如 *KerasPolicy*）。用于在忽略训练数据记忆转折时测试集合的预测能力。

train（*resource_name = None*，*model_path = None*，*remove_duplicates = True*，*** kwargs* ）[source]

使用来自文件的对话数据来训练策略/策略集成

train_online（*resource_name = None*，*input_channel = None*，*model_path = None*，*** kwargs* ）[source]

在设定的策略/集合上运行在线训练课程。

这些策略将使用来自 *文件名* 的数据进行预训练。之后，模型将接受来自输入通道的对话训练。在对话期间，可以改变代理人的注释和状态以纠正错误的行为。

visualize（*resource_name*，*output_file*，*max_history*，*nlu_training_data = None*，*fontsize = 12* ）[source]

事件

对话系统能够处理和支持的所有事件的列表。

一个事件可以被它的 `type_name` 属性引用（例如，当在一个 http 回调中返回事件时）。

class rasa_core.events.ActionExecuted（*action_name*，*timestamp = None*）[source]

基地： `rasa_core.events.Event`

操作描述了所采取的行动及其结果。

它包括一个动作和一系列事件。操作将被追加到最新 `Turn` 的 `Tracker.turns`。

apply_to(*tracker*)[source]

as_dict()[source]

as_story_string()[source]

type_name= *u'action'*

class rasa_core.events.ActionReverted（*timestamp = None*）[source]

基地： `rasa_core.events.Event`

机器人撤销其最后的行动。

不应该在实际的用户交互中使用，主要是用于火车。作为副作用，`Tracker` 最后一回合被删除。

apply_to(*tracker*)[source]

as_story_string()[source]

type_name= *u'undo'*

class rasa_core.events.AllSlotsReset（*timestamp = None*）[source]

基地： `rasa_core.events.Event`

对话应该重新开始并清除历史记录。

作为副作用，`Tracker` 将被重新初始化。

apply_to(*tracker*)[source]

as_story_string()[source]

type_name= *u'reset_slots'*

class rasa_core.events.BotUttered（*text = None*，*data = None*，*timestamp = None*）[source]

基地： `rasa_core.events.Event`

机器人已经向用户说了些什么。

这个课程不用于故事训练，因为它包含在

`ActionExecuted`类。一个条目是在 `Tracker`。

apply_to(*tracker*)[source]

as_dict()[source]

as_story_string()[source]

*static***empty**()[source]

type_name= *u'bot'*

class **rasa_core.events.ConversationPaused**（*timestamp = None*）[source]

基地： `rasa_core.events.Event`

忽略来自用户的消息让人类接管。

作为副作用 `Tracker`的 `paused`属性将被设置为 `True`。

apply_to(*tracker*)[source]

as_story_string()[source]

type_name= *u'pause'*

class **rasa_core.events.ConversationResumed**（*timestamp = None*）[source]

基地： `rasa_core.events.Event`

Bot 接管对话。

反过来 `PauseConversation`。作为副作用 `Tracker`的 `paused`属性将被设置为 `False`。

apply_to(*tracker*)[source]

as_story_string()[source]

type_name= *u'resume'*

class **rasa_core.events.Event**（*timestamp = None*）[source]

基地： `object`

事件是以下情况之一： - 用户对机器人说过的话（开始新的一轮） - 话题已被设置 - 机器人已采取行动

事件由 Tracker 的更新方法记录。这将更新转动列表，以便通过消耗转动列表来恢复当前状态。

apply_to(*tracker*)[source]

as_dict()[source]

as_story_string()[source]

*static***from_parameters**(*parameters*, *default=None*)[source]

*static***from_story_string**(*event_name*, *parameters*, *default=None*)[source]

*static***resolve_by_type**(*type_name*, *default=None*)[source]

按类型名称返回一个 slots 类。

type_name= *u'event'*

class **rasa_core.events.ReminderScheduled**(*action_name*, *trigger_date_time*, *name = None*, *kill_on_user_message = True*, *timestamp = None*) [source]

基地： `rasa_core.events.Event`

允许对操作执行进行异步调度。

作为副作用，消息处理器将安排在触发日期运行的操作。

as_dict () [source]

as_story_string () [source]

type_name= *u'reminder'*

class **rasa_core.events.Restarted**（*timestamp = None* ） [source]

基地： `rasa_core.events.Event`

对话应该重新开始并清除历史记录。

作为副作用， `Tracker` 将被重新初始化。

apply_to（*tracker*） [source]

as_story_string () [source]

type_name= *u'restart'*

class **rasa_core.events.SlotSet**(*key*, *value = None*, *timestamp = None*)[source]

基地： `rasa_core.events.Event`

用户已经为 a 的值指定了他们的偏好 `slot`。

作为副作用 `Tracker` 的插槽将更新，以便 `tracker.slots[key]=value`。

apply_to（*tracker*） [source]

as_dict（） [source]

as_story_string（） [source]

type_name= *u'slot'*

class **rasa_core.events.StoryExported**（*path = None*，*timestamp = None* ） [source]

基地： `rasa_core.events.Event`

故事应该被转储到一个文件中。

apply_to（*tracker*） [source]

as_story_string（） [source]

type_name= *u'export'*

class **rasa_core.events.TopicSet**（*topic*，*timestamp = None* ） [source]

基地： `rasa_core.events.Event`

谈话的话题已经改变。

作为一种副作用，self.topic 将被推向 `Tracker.topic_stack`。

apply_to（*tracker*） [source]

as_dict（） [source]

as_story_string（） [source]

type_name= *u'topic'*

class **rasa_core.events.UserUtteranceReverted**（*timestamp = None* ） [source]

基地： `rasa_core.events.Event`

机器人撤销其最后的行动。

不应该在实际的用户交互中使用，主要是用于火车。作为副作用，`Tracker` 最后一回合被删除。

apply_to（*tracker*）[source]

as_story_string（）[source]

type_name= *u'rewind'*

class rasa_core.events.UserUttered（*text*，*intent = None*，*entities = None*，*parse_data = None*，*timestamp = None* ）[source]

基地： rasa_core.events.Event

用户对机器人说了些什么。

作为一个副作用，一个新的 Turn 将在创造 Tracker。

apply_to（*tracker*）[source]

as_dict（）[source]

as_story_string（）[source]

*static*empty（）[source]

type_name= *u'user'*

rasa_core.events.deserialise_events（*serialized_events* ）[source]

将字典列表转换为相应事件的列表。

示例格式：

[{“event”：“set_slot”，“value”：5，“name”：“my_slot”}]

特征提取

为了将机器学习算法应用于对话式 AI，我们需要建立对话的矢量表示。

我们使用监督式学习常用的符号，其中是形状矩阵，是包含目标类别标签的一维长度数组。`X, yX(num_data_points, data_dimension)ynum_data_points`

目标标签对应于机器人采取的动作。如果域定义可能，那么标签 0 表示问候语，1 表示听。`actions[ActionGreet, ActionListen]`

行中的行 `x` 对应于采取行动之前的对话状态。

对单一状态进行归纳是这样工作的：跟踪器提供了一个 `active_features` 包括：

- 最后一个动作是什么（例如 `prev_action_listen`）
- 表示意图和实体的特征，如果这是一个回合中的第一个状态，例如它是解析用户消息后我们将要采取的第一个行动。（例如）`[intent_restaurant_search, entity_cuisine]`
- 指示当前定义哪些插槽的功能，例如，`slot_location` 如果用户先前提及了他们正在搜索餐馆的区域。
- 指示存储在槽中的任何 API 调用的结果的特征，例如，`slot_matches`

所有这些功能都以二进制向量表示，只表示它们是否存在。

例如 `[0 0 1 1 0 1 ...]`

要从矢量中恢复特征包 `vec`，可以调用 `domain.reverse_binary_encoded_features(vec)`。这对调试非常有用。

历史

包含更多的历史记录通常比内存中的当前状态更有用。该参数 `max_history` 定义了在每个行中定义了多少个状态 `x`。

因此，上面的说法 `x` 是 2D 实际上是假的，它已经形成。对于大多数算法你想要一个平坦的特征向量，所以你将不得不重塑这个。

```
(num_states, max_history, num_features)(num_states, max_history * num_features)
```

解释器

解释文本的工作大部分在 Rasa Core 的范围之外。要将文本转换为结构化数据，您可以使用 Rasa NLU 或像 wit.ai 这样的云服务。如果您的机器人使用按钮点击或其他非自然语言输入，则根本不需要任何内容 。您可以定义自己的 `Interpreter` 子类，它可以执行您可能需要的任何自定义逻辑。你可以看看这个 `RegexInterpreter` 类。

要使用除 Rasa NLU 之外的其他类，只需实现一个子类，`Interpreter` 该子类具有一个 `parse(message)` 只接受一个字符串参数的方法，并以下列格式返回一个字典：

```
{
  "text": "show me chinese restaurants",
  "intent": {
    "name": "restaurant_search",
    "confidence": 1.0
  }
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine"
    }
  ]
}
```

固定意图和实体输入

有时候，你想确保一条消息被视为包含已定义实体的固定意图。为此，您可以使用标记格式指定消息，而不是使用消息文本。

不要发送像希望被正确分类的消息，你可以绕过 NLU 并直接发送一个消息。Rasa Core 将把这个传入的消息看作是一个具有意图的正常消息和具有值的实体。

`Hello I am Rasa/greet{"name": "Rasa"}greetnameRasa`

如果你想指定一个输入字符串，它包含你可以使用的同一类型的多个实体值

```
/ add_to_shopping_list { “item”: [ “milk”, “salt” ]}
```

这与消息相对应。`"I want to add milk and salt to my list"`

自定义策略

这 `Policy` 是你的机器人的核心，它真的只有一个重要的方法：

```
def predict_action_probabilities(self, tracker, domain):
    # type: (DialogueStateTracker, Domain) -> List[float]

    return []
```

这将使用会话的当前状态（由跟踪器提供）来选择要采取的下一个操作。如果你需要的话，该域名在那里，但只有一些策略类型可以使用它。返回的数组包含接下来要执行的每个动作的概率。最有可能的动作将被执行。

我们来看一个自定义策略的简单例子：

```
from rasa_core.policies import Policy
from rasa_core.actions.action import ACTION_LISTEN_NAME
from rasa_core import utils
import numpy as np

class SimplePolicy(Policy):
    def predict_action_probabilities(self, tracker, domain):
        responses = {"greet": 3}

        if tracker.latest_action_name == ACTION_LISTEN_NAME:
            key = tracker.latest_message.intent["name"]
            action = responses[key] if key in responses else 2
            return utils.one_hot(action, domain.num_actions)
        else:
            return np.zeros(domain.num_actions)
```

这个怎么用？ 当控制器处理来自用户的消息时，它将继续询问下一个最可能使用的动作 `predict_action_probabilities`。机器人然后执行该动作，直到收到 `ActionListen` 指令。这打破了循环，并使机器人等待进一步的指示。

在伪代码中，`SimplePolicy` 上面所做的是：

```
-> a new message has come in

if we were previously listening:
    return a canned response
else:
    we must have just said something, so let's Listen again
```

请注意，策略本身是无状态的，并且所有状态都由 `tracker` 对象携带。

从故事创造策略

像上面的 SimplePolicy 一样编写规则并不是构建 bot 的好方法，它很快就会变得杂乱，很难调试。如果你找到了 Rasa Core，很可能你已经尝试了这种方法，并且正在寻找更好的东西。接下来的一步就是使用我们的故事框架，通过给出一些示例对话来构建策略。我们不会使用机器学习，我们只会创建一个记录这些故事的策略。

我们可以使用 `MemoizationPolicy` 和 `PolicyTrainer` 类来做到这一点。

这是 `PolicyTrainer` 类：

```
class PolicyTrainer(object):
    def __init__(self, ensemble, domain, featurizer):
        self.domain = domain
        self.ensemble = ensemble
        self.featurizer = featurizer

    def train(self, filename=None, max_history=3,
              augmentation_factor=20, max_training_samples=None,
              max_number_of_trackers=2000, remove_duplicates=True, **kwargs):
        """Trains a policy on a domain using training data from a file.

        :param augmentation_factor: how many stories should be created by
                                    randomly concatenating stories
        :param filename: story file containing the training conversations
        :param max_history: number of past actions to consider for the
                            prediction of the next action
        :param max_training_samples: specifies how many training samples to
                                     train on - `None` to use all examples
        :param max_number_of_trackers: limits the tracker generation during
                                       story file parsing - `None` for unlimited
        :param remove_duplicates: remove duplicates from the training set before
                                  training
        :param kwargs: additional arguments passed to the underlying ML trainer
                       (e.g. keras parameters)
        :return: trained policy
        """

        logger.debug("Policy trainer got kwargs: {}".format(kwargs))
        check_domain_sanity(self.domain)

        training_data = self._prepare_training_data(filename, max_history,
                                                    augmentation_factor,
                                                    max_training_samples,
```

```
                                    max_number_of_trackers,
                                    remove_duplicates)

        self.ensemble.train(training_data, self.domain, self.featurizer, **kwargs)

    def _prepare_training_data(self, filename, max_history, augmentation_factor,
                               max_training_samples=None,
                               max_number_of_trackers=2000,
                               remove_duplicates=True):
        """Reads training data from file and prepares it for the training."""

        from rasa_core.training import extract_training_data_from_file

        if filename:
            training_data = extract_training_data_from_file(
                    filename,
                    self.domain,
                    self.featurizer,
                    interpreter=RegexInterpreter(),
                    augmentation_factor=augmentation_factor,
                    max_history=max_history,
                    remove_duplicates=remove_duplicates,
                    max_number_of_trackers=max_number_of_trackers)
            if max_training_samples is not None:
                training_data.limit_training_data_to(max_training_samples)
            return training_data
        else:
            return DialogueTrainingData.empty(self.domain)
```

该 `train()` 方法的作用如下：

1. 从文件中读取故事
2. 从这些故事中创建所有可能的对话
3. 创建以下变量：
   a. `y` - 表示在对话中采取的所有动作的 1D 数组
   b. `X` - 一个 2D 数组，其中每行表示执行操作时跟踪器的状态
4. 调用策略的 `train()` 方法来从这些 状态行为对创建策略（不要介意它只是一组策略 - 例如，您可以组合多个策略并使用集合一次性训练它们） `X, yensemble`

注意

事实上，当前 `X` 面的 `max_history` 操作被采用时，行中描述了跟踪器的状态。有关更多详细信息，请参阅 Featurization。

对于这个 `MemoizationPolicy`，`train()`方法只是记住故事中采取的行动，以便当你的机器人遇到相同的情况时，它会做出你想要的决定。

推广到新的对话

故事数据格式为您提供了一种紧凑的方式，可以毫不费力地描述大量可能的对话。但人类无限创意，你永远不可能希望以编程的方式来描述*每一种*可能的对话。即使你可以，它可能不适合内存:)

那么，我们如何创建一个即使在你没有想到的情景中也表现良好的策略？我们将尝试通过创建基于机器学习的策略来实现这一概括。

您可以使用任何您喜欢的机器学习库来训练您的策略。Rasa 附带的一个实现是`KerasPolicy`使用 Keras 作为机器学习库来训练对话模型。这些基类已经实现了持久化和重新加载模型的逻辑。

默认情况下，每个都会训练线性模型以适应数据。`X, y`

模型在这里定义：

```
def model_architecture(self, num_features, num_actions, max_history_len):
    """Build a keras model and return a compiled model.

    :param max_history_len: The maximum number of historical
                            turns used to decide on next action
    """
    from keras.layers import LSTM, Activation, Masking, Dense
    from keras.models import Sequential

    n_hidden = 32  # Neural Net and training params
    batch_shape = (None, max_history_len, num_features)
    # Build Model
    model = Sequential()
    model.add(Masking(-1, batch_input_shape=batch_shape))
    model.add(LSTM(n_hidden, batch_input_shape=batch_shape, dropout=0.2))
    model.add(Dense(input_dim=n_hidden, units=num_actions))
    model.add(Activation('softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer='rmsprop',
                  metrics=['accuracy'])
```

```
        logger.debug(model.summary())
        return model
```

并在这里进行训练：

```
    def train(self, training_data, domain, **kwargs):
        # type: (DialogueTrainingData, Domain, **Any) -> None
        self.model = self.model_architecture(domain.num_features,
                                              domain.num_actions,
                                              training_data.max_history())
        shuffled_X, shuffled_y = training_data.shuffled(domain)

        validation_split = kwargs.get("validation_split", 0.0)
        logger.info("Fitting model with {} total samples and a validation "
                    "split of {}".format(training_data.num_examples(),
                                         validation_split))
        self.model.fit(shuffled_X, shuffled_y, **kwargs)
        self.current_epoch = kwargs.get("epochs", 10)
        logger.info("Done fitting keras policy model")
```

您可以通过重写这些方法来实现您选择的模型。

跟踪对话状态

这 `DialogueStateTracker` 是跟踪对话的状态对象。跟踪器应该更新的唯一方法就是传递 `events` 给 `update` 方法。例如：

```
>>> from rasa_core.trackers import DialogueStateTracker
>>> from rasa_core.slots import TextSlot
>>> from rasa_core.events import SlotSet

>>> tracker = DialogueStateTracker("default", slots=[TextSlot("cuisine")])
>>> print(tracker.slots)
{u'cuisine': <TextSlot(cuisine: None)>}
>>> tracker.update(SlotSet("cuisine","Mexican"))
>>> print(tracker.slots)
{u'cuisine': <TextSlot(cuisine: Mexican)>}
```

事件 API 文档中记录了整套事件。

坚持追踪者：

当你在生产中运行你的机器人时，你希望你的应用程序是无状态的。例如，每次重新启动正在运行的进程时，您都不想丢失每个对话的跟踪。这就是为什么 Rasa 在关键价值商店坚持追踪者的原因。对于测试来说，这 `InMemoryTrackerStore` 已经足够了，但是在生产中，您希望 `RedisTrackerStore` 在重新启动应用程序后使用它来进行恢复。`TrackerStore` 为您选择的持久化工具定义自定义子类很简单。

序列化

Rasa 创建一个 `dialogue` 串行化对象，而不是沉浸在跟踪器对象的最终状态。对话是以前 `N` 对话轮流的完整记录。要返回到当前的对话状态，我们迭代轮流并记录每个事件。

我们使用 `jsonpickle` 库来连载这些对话框。下面是一个简单的对话示例，因为它将存储在 TrackerStore 中：

```
{
  "py/object":"rasa_core.conversation.Dialogue",
  "events":[
    {
      "py/object": "rasa_core.events.UserUttered",
      "entities": [],
      "intent": {
        "name": "greet",
        "confidence": 1.0
      },
      "text": "/greet"
    },
    {
      "py/object": "rasa_core.events.ActionExecuted",
      "action_name": "utter_greet",
      "unpredictable": false
    }
  ],
  "name":"hello_world"
}
```

迁移指南

此页面包含有关主要版本之间更改以及如何从一个版本迁移到另一个版本的信息。

0.7.x 至 0.8.0

- Facebook 连接器的凭证已更改。而不是提供

```
# OLD FORMAT
verify: "rasa-bot"
secret: "3e34709d01ea89032asdebfe5a74518"
page-tokens:
1730621093913654: "EAAbHPa7H9rEBAAuFk4Q3gPKbDedQnx4djJJ1JmQ7CAqO4iJKrQcNT0wtD"
```

你现在应该像这样传递配置参数：

```
# NEW FORMAT
verify: "rasa-bot"
secret: "3e34709d01ea89032asdebfe5a74518"
page-access-token: "EAAbHPa7H9rEBAAuFk4Q3gPKbDedQnx4djJJ1JmQ7CAqO4iJKrQcNT0wtD"
```

正如你所看到的，新的 facebook 连接器只支持一个页面。连接器的代码参数发生了相同的变化，应更改为：

```
from rasa_core.channels.facebook import FacebookInput

FacebookInput(
      credentials.get("verify"),
```

```
        credentials.get("secret"),
        credentials.get("page-access-token"))
```

- 从改变故事的文件格式，以（旧格式仍然支持，但反对使用）。而不是写作 `* _intent_greet[name=Rasa]* intent_greet{"name": "Rasa"}`

```
## story_07715946                    <!-- name of the story - just for debugging -->
* _greet
   - action_ask_howcanhelp
* _inform[location=rome,price=cheap]
   - action_on_it                    <!-- user utterance, in format _intent[entities]
-->
   - action_ask_cuisine
```

新的格式如下所示：

```
## story_07715946                    <!-- name of the story - just for debugging -->
* greet
   - action_ask_howcanhelp
* inform{"location": "rome", "price": "cheap"}
   - action_on_it                    <!-- user utterance, in format _intent[entities]
-->
   - action_ask_cuisine
```

更改日志

这个项目的所有显着变化都将记录在这个文件中。该项目遵循从版本 0.2.0 开始的语义版本控制。

[Unreleased 0.9.0.aX] - master

注意

该版本尚未发布，正在积极开发中。

添加

变

删除

固定

[0.8.5] - 2018-03-19

固定

- 更新谷歌分析文档调查代码

[0.8.4] - 2018-03-14

固定

- 引脚 `pykwalify<=1.6.0` 作为更新来 `1.6.1` 打破兼容性

[0.8.3] - 2018-02-28

固定

- pin `fbmessenger` 版本以避免重大更新

[0.8.2] - 2018-02-13

添加

- 脚本重新加载倾销的跟踪器状态，并在存储的对话结束时继续对话

变

- 依赖关系的小更新

固定

- 提醒事件的固定日期时间序列化

[0.8.1] - 2018-02-01

固定

- 删除了 deque 以支持 python 3.5
- 对教程的文档改进
- 序列化 `ReminderScheduled` 事件的日期时间值

[0.8.0] - 2018-01-30

这是一个主要的版本变化。请务必阅读文档中的“ 迁移指南”，以获取有关如何更新现有项目的建议。

添加

- `--debug` 和 `--verbose` 脚本标记（train.py，run.py，server.py）来设置日志级别
- 在使用检查点时支持故事周期
- 添加了一个新的机器学习策略 *SklearnPolicy*, 该策略使用 sklearn 分类器来预测动作（默认为逻辑回归）
- 警告如果行动在使用它在模型训练过的任何故事中从未发出的模型时发出事件
- 支持事件推送和端点从服务器检索跟踪器状态
- 时间戳记到每个事件
- 添加了一个 Slack 频道，允许 Rasa Core 通过 Slack 应用进行通信
- 添加了一个允许 Rasa Core 通过电报机器人进行通信的电报通道

变

- 重写整个 FB 连接器：用 fbmessenger 替换 pymessenger 库
- 故事文件的话语格式从原来的格式 变为（旧格式仍然受支持，但已弃用）
  `* _intent_greet[name=Rasa]* intent_greet{"name": "Rasa"}`
- 在模型持久化期间在域中保留动作名称
- 通过不使用 miniconda 来提高特拉维斯速度
- 如果话语模板包含无法填充的变量，则不会失败并显示一条有用的错误消息
- 域不会在未知操作上失败，但会发出警告。这是为了支持从最近的会话中读取日志，如果最近从域中删除了一个动作

固定

- 用检查点正确评估故事
- 用检查点对故事进行适当的可视化
- 固定浮动槽最小值最大值处理
- 固定的非整数特征解码，例如用于记忆策略
- 在启动 Rasa Core 服务器时正确登录到指定的文件
- 从跟踪器商店加载跟踪器后，正确计算上次重置事件的偏移量
- UserUtteranceReverted 操作错误地触发了要重播的操作

[0.7.9] - 2017-11-29

固定

- 使用 Networkx 版本 2.x 进行可视化
- 在解析故事文件时添加关于失败意图行的输出

[0.7.8] - 2017-11-27

固定

- Pypi 自述渲染

[0.7.7] - 2017-11-24

添加

- 将机器人语音记录到跟踪器

固定

- README 中的文档改进
- 更名为 rasa 核心服务器的解释器参数

[0.7.6] - 2017-11-15

固定

- moodbot 示例在文档中训练命令

[0.7.5] - 2017-11-14

变

- “sender_id”（和“DEFAULT_SENDER_ID”）关键字一致性问题＃56

固定

- 改进的 moodbot 示例 - 更多的 nlu 示例以及更好的对话模型拟合

[0.7.4] - 2017-11-09

变

- 添加方法来跟踪器来检索最新的实体＃68

[0.7.3] - 2017-10-31

添加

- 参数在渲染故事可视化时指定字体大小

固定

- 固定的故事可视化文档

[0.7.2] - 2017-10-30

添加

- 增加了 Facebook bot 例子
- 增加了对条件检查点的支持。如果设置了特定的插槽，则可以限制检查点仅允许使用该检查点。详情请参阅文档
- 域 yaml 中的话语模板支持按钮和图像
- 验证域 yaml 并在无效文件上引发异常
- `run` 脚本来加载模型并处理来自输入通道的消息

变

- 标准 keras 模型中的小辍学率，以减少对确切意图的依赖
- 很多文档的改进

固定

- 如果操作监听未被确认，则修复 http 错误。 # 42

[0.7.1] - 2017-10-06

固定

- 问题与重新启动事件。他们创造了错误的历史导致了错误的预测

[0.7.0] - 2017-10-04

添加

- 支持 Rasa Core 作为远程操作执行的服务器

变

- 切换到最大码行长度 80
- 删除操作 ID - 使用 `action.name()`。如果一个动作实现重写名称，它应该包含 `action_` 前缀（因为它不会自动添加）
- 改名 `rasa_dm.util` 为 `rasa_dm.utils`
- 将整个包重命名为 `rasa_core` （如此 `rasa_dm` 离开 !）
- 重命名 `Reminder` 属性 `id` 为 `name`
- 大量的文档改进。docs 现在位于 https://core.rasa.ai
- 在将记忆转化为持久性时使用散列 - 要求重新训练所有使用此版本之前版本的模型
- 改变了 `agent.handle_message(...)` 界面以方便使用

[0.6.0] - 2017-08-27

添加

- 支持多项策略（例如同时进行一项备忘录和 Keras 策略）
- 从 yaml 文件加载域，而不是用 python 代码定义它们
- 添加了一个 api 图层（调用 `Agent`），供您使用 95％的要做的事情（训练，持久性，加载模型）
- 支持提醒

变

- 大量的代码库重构

[0.5.0] - 2017-06-18

添加

- `ScoringPolicy` 添加到策略实施（比标准默认策略更严格）
- `RasaNLUInterpreter` 在 dm 中运行一个 nlu 实例（而不是使用 http 接口）
- 更多的测试

变

- `UserUtterance` 现在持有来自 nlu 的完整解析数据（例如，访问除实体或意图以外的属性）
- `Turn` 有一个引用 `UserUtterance` 而不是直接存储意图和实体（允许访问其他数据）
- 输出通道的简化界面
- DefaultPolicy 中的操作顺序 `possible_actions`（`ActionListen` 现在总是有索引 0）

固定

- `RedisTrackerStore` 检查跟踪器在访问之前是否存储（否则 `None` 会引发访问异常）
- `RegexInterpreter` 检查正则表达式实际上是否与消息匹配，而不是假定它总是这样做
- `str` 实施所有事件
- `Controller` 可以在没有输入通道的情况下启动（例如，需要手动将消息输入到队列中）

[0.2.0] - 2017-05-18

首先发布的版本。