# Проектирование хранилищ больших объемов данных

Занятие №9. Spotify Luigi

# Содержание

# Запуск Spark-приложений

# SparkSession

```
1.  import pyspark
2.
3.  spark = pyspark.sql.SparkSession.builder\
4.      .master('yarn-client')\ # cluster-manager, для standalone ‘local[2]’
5.      .appName('app_name')\ # название приложения
6.      .config('spark.executor.instances', '2')\ # число исполнителей
7.      .config('spark.executor.memory', '1G')\ # объем памяти доступной
    исполнителям
8.      .config('spark.executor.cores', '2')\ # число ядер, доступных исполнителю
9.      .config('spark.dynamicAllocation.enabled', 'false')\
10.     .config('spark.driver.memory', '1G')\ # объем памяти, доступной драйверу
11.     .getOrCreate()
```

# Spark-submit

```
export JAVA_HOME=/usr/java/jdk1.8.0_191/jre
export SPARK_HOME=/usr/hdp/current/spark2-client
export HADOOP_CONF_DIR=/usr/hdp/current/hadoop-client/etc/hadoop

spark-submit \
--master yarn \
--name testmyapp \
--deploy-mode cluster \
--num-executors=1 \
--executor-memory=2G \
--executor-cores=2 \
--driver-memory=1G \
--conf "spark.dynamicAllocation.enabled=false" \ ~/test_sub.py
```

https://spark.apache.org/docs/2.3.2/submitting-applications.html
https://spark.apache.org/docs/2.3.2/configuration.html

# Cron

# Cron

**crond** – демон для исполнения команд по расписанию

Расписание задается при помощи т.н. “crontab’ов”, для работы с которыми имеется утилита crontab (с флагом -e ваш конфиг в редакторе)

Синтаксис:

```
0 0 * * * cmd
| | | | |
| | | | +- day of week
| | | +--- month
| | +----- day of month
| +------- hour
+--------- minute
```

https://crontab.guru/

# Luigi

# Spotify Luigi

Фреймворк с открытым исходным кодом, предназначенный для построения сложных “пайплайнов”. Берет на себя разрешение зависимостей, управление процессами, визуализацию, обработку ошибок, интеграцию с командной строкой и многое другое.


https://github.com/spotify/luigi

git clone https://github.com/Ptomine/luigi.git

cd luigi/examples/

PYTHONPATH="." luigi --module module_name TaskClassName  --local-scheduler \
--param-name param-value –bool-param

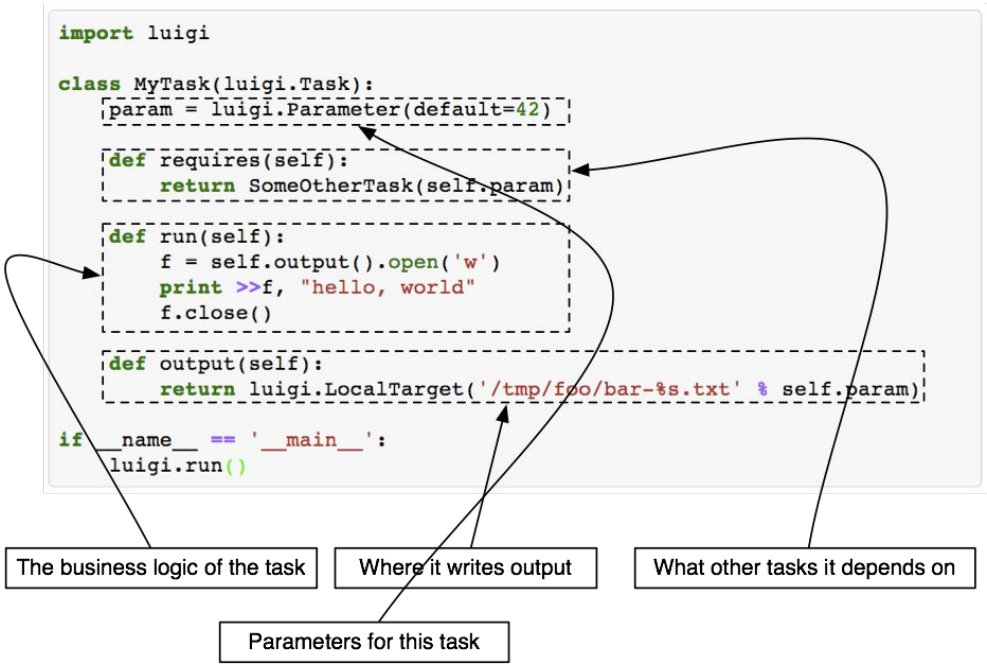
# Основные сущности

- Target - “цель”, обеспечивает связь с объектом ФС, записью в таблице БД и т.д., которая будет являться результатом работы части пайплайна. При наследовании мы обязаны переопределить метод exists()
- Task - обеспечивает выполнение логики. Для управления поведением можем переопределять методы output(), requires() и run()

*Не рекомендуется создавать Task’и, имеющие более одного Target’а, для простоты поддержания атомарности операций (если она нужна). Лучше создать множество Task’ов, имеющих по одному Target’у*

Task Visualiser - http://vk-edu-s202346b22c11-head-0.mcs.local:8082

# Task




requires() - используется для описания других **Task'ов(!)**, от которых зависит текущий. Может вернуть как объект, так и Iterable (а значит и в т.ч. dict)

output() - определяем Target'ы

run() - исполняемый код. Для построения динамических зависимостей допустимо yield'ить другие Task'и из этого метода

input() - обертка над requires, вернет output'ы всех Task'ов, от которых зависит текущий

**WrapperTask** - таск который сам по себе ничего не делает, а только порождает зависимости. Полезен для изоляции логики пайплайна от логики его запуска, для запуска множества одинаковых тасок с разными параметрами и т.д.

# Parameter

priority - числовой параметр, влияющий на приоритет выполнения Task'а. Чем больше, тем раньше

*На самом деле на порядок выполнения влияет не только priority, но и зависимости, они влияют сильнее. Например: Task A с priority=1000, но имеющий отложенные зависимости и Task B с priority=1 не имеющий отложенных зависимостей, Task B будет взят в работу первым.*

При добавлении своих параметров в наследников абстрактного класса Task, Luigi сам добавит нужные поля в конструктор, а также возьмет на себя обработку аргументов командной строки с их приведением в необходимые типы Python

Таски одного класса с одинаковыми значениями параметра на самом деле будут являться одним инстансом. Такого же эффекта можно добиться и при отличных параметров, для этого нужно при добавлении параметра в класс передать аргумент significant=false:

```
foo = luigi.Parameter(significant=False)
```

*Если параметр содержит в названии "_" при запуске из командной строки их нужно заменить на "-"*

# Configuration

Файлы конфигурации имеют структуру:

```
[section]
parameter: value

[MyTaskClassName]
source_path: /path/to/data

[MySecondClassName]
execute: false
```

ConfigParser будет искать нужную секцию в файлах в следующем порядке:

1. /etc/luigi/luigi.cfg - как правило это глобальный файл конфигурации, использовать стоит соответствующим образом
2. luigi.cfg - файл в текущей директории, удобно использовать для конфигурации конкретного пайплайна
3. путь в переменной окружения LUIGI_CONFIG_PATH - удобно использовать для конфигурации какого-то проекта или нескольких пайплайнов, как нечто среднее между глобальным конфигом и конфигом пайплайна

# Бонус

Task.clone() - когда нужно породить таск-зависимость, в котором все или большинство параметров те же что и у родителя. Отличные передаются аргументами в метод.

Task.complete() - на самом деле именно этот метод говорит sheduler’у, завершен ли таск. По умолчанию вернет True если существуют все output’ы, но иногда может быть полезным его переопределить

Task.on_success() и Task.on_failure() - кастомизация поведения при успешном завершении и обработки ошибок (on_failure принимает exception)

# Спасибо
# за внимание!