# 1．技术栈分析以及用途

Eggroll 技术栈由 scala、java 组成。

* scala分布式集群构建
* java用于包装grpc数据传输

eggroll-all项目（4个）

* eggroll-core 核心包（java和sclala）
* eggroll-roll-frame grpc消息类型（sclala）
* eggroll-roll-pair 分布式计算 （sclala）
* eggroll-roll-site 代理网关 外网数据交换服务（java和sclala）

# 2．类图之间的依赖



绿色为python 浅蓝scala 深蓝java 函数调用流程

# 3．业务调用流程图



Reduce函数调用流程



初始化流程

init()初始化设置工作模式集群或单机，设置处模式理eggroll或spark，build\_session()创建session，FateSessionImpl()内部类初始化\_init\_()调用scala RollPairContext类，load()方法集群创建数据空间，底层通过grpc传输

# 4．操作API

## Session API

### 函数列表

|  |  |
| --- | --- |
| 函数 | 描述 |
| [**cleanup**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.cleanup)(name, namespace[, persistent]) | 销毁表函数 |
| [**generateUniqueId**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.generateUniqueId)() | 每次调用时生成一个唯一的ID |
| [**get\_data\_table**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.get_data_table)(name, namespace) | 按表名和表名空间返回数据表实例 |
| [**get\_data\_table\_meta**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.get_data_table_meta)(key, data\_table\_name, …) | 从data\_table\_name的表和命名空间的data\_table\_namespace关联的元表中按键获取元键。 |
| [**get\_data\_table\_metas**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.get_data_table_metas)(data\_table\_name, …) | 从data\_table\_name的表和命名空间的data\_table\_namespace关联的meta表中获取meta。 |
| [**get\_session\_id**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.get_session_id)() | 返回sessionId |
| [**init**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.init)([job\_id, eggroll\_version, set\_log\_dir]) | 初始化会话，应先调用 |
| [**parallelize**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.parallelize)(\*args, \*\*kwargs) | 将现有的数据转换为表。 |
| [**save\_data**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.save_data)(kv\_data, name, namespace[, …]) | 将数据保存到表中，可选添加版本。 |
| [**save\_data\_table\_meta**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.save_data_table_meta)(kv, data\_table\_name, …) | 将元数据（以kv为单位）保存到与名为data\_table\_name和命名空间为data\_table\_namespace的表关联的meta table。 |
| [**stop**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.stop)() | 停止会话，清理与此会话关联的所有表。 |
| [**table**](https://fate.readthedocs.io/en/latest/doc/api/session_api.html#arch.api.session.table)(\*args, \*\*kwargs) | 加载现有表 |

#### cleanup(name, namespace[, persistent])

**cleanup(*name*, *namespace*, *persistent=False*)**

**参数：**

* name（string）–要清理的表名。这里可以使用通配符。
* namespace（string）–要清理的表命名空间。这需要完全匹配。
* persistent（boolean）–删除表的位置，从持久存储中删除True，从临时存储中删除False。

**返回：**

无

**返回类型：**

None

**Examples**

**>>> from** **arch.api** **import** session

**>>>** session.cleanup('foo\*', 'bar', persistent=**True**)

#### get\_data\_table(name, namespace)

**参数：**

* key（string）–关联的键。
* data\_table\_name（string）–此数据表的表名
* data\_table\_namespace（string）–此数据表的表名

**返回：**

与提供的键关联的对象

**返回类型：**

any

**Examples：**

**>>> from** **arch.api** **import** session

**>>>** session.get\_data\_table\_meta("model\_id", "meta", "readme") *# a\_id*

#### get\_data\_table\_metas(data\_table\_name, data\_table\_namespace)

**参数：**

* data\_table\_name（string）–此数据表的表名
* data\_table\_namespace（string）–此数据表的表名

**返回：**

Metas

**返回类型：**

dict

**Examples：**

**>>> from** **arch.api** **import** session

**>>>** session.get\_data\_table\_metas("meta", "readme") *# {'model\_id': 'a\_id', 'used\_framework': 'fate'}*

#### init([job\_id, eggroll\_version, set\_log\_dir])

**init(job\_id=None, mode: int = <WorkMode.STANDALONE: 0>, backend: int = <Backend.EGGROLL: 0>, persistent\_engine: str = 'LMDB', eggroll\_version=None, set\_log\_dir=True, options: dict = None)**

**参数：**

* job\_id（string）–此运行时的作业id和默认表命名空间。
* mode（WorkMode）-模式（工作模式）

设置工作模式，

Standalone（单机版）：工作模式 WorkMode.STANDALONE or 0

Cluster（集群）：WorkMode.CLUSTER or 1

* backend (Backend) –后端

设置计算后端，

* eggroll: Backend.EGGROLL or 0
* spark: Backend.SAPRK or 1
* options (None or dict) –其他选项

**返回：**

没有返回

**返回类型：**

None

**Examples：**

**>>> from** **arch.api** **import** session, WorkMode, Backend

**>>>** session.init("a\_job\_id", WorkMode.Standalone, Backend.EGGROLL)

#### parallelize(\*args, \*\*kwargs)

**parallelize(*\*args*, *\*\*kwargs*) → [arch.api.base.table.Table](https://fate.readthedocs.io/en/latest/doc/api/table_api.html" \l "arch.api.base.table.Table" \o "arch.api.base.table.Table)**

**参数：**

* data（Iterable）–要放入的数据。
* Include\_key（boolean）–将数据并行化到表中时是否包含key。
* name（string）–结果表的表名。当不使用时，将生成一个默认表名
* partition（int）–并行化数据时的分区数。
* namespace（string）–结果表的表名称空间。未使用作业时将使用作业\_id。
* create\_if\_missing（boolean）–未实现。将始终创建表。
* error\_if\_exist（boolean）-如果不存在，则返回布尔值。如果已经存在，则不会抛出错误。
* chunk\_size（int）–将数据并行到表中时的批处理大小。
* in\_place\_computing（boolean）–是否启用就地计算。

**返回：**

由并行化数据组成的表

**返回类型：**

Table

**Examples：**

**>>> from** **arch.api** **import** session

**>>>** table = session.parallelize(range(10), in\_place\_computing=**True**)

#### save\_data(kv\_data, name, namespace[, …])

**save\_data(*kv\_data:* *Iterable*, *name*, *namespace*, *partition=1*, *persistent:* *bool* *=* *True*, *create\_if\_missing=True*, *error\_if\_exist=False*, *in\_version:* *bool* *=* *False*, *version\_log=None*)**

**参数：**

* kv\_data（Iterable）–要保存的数据
* name (string)）–表名
* namespace（string）–表名称空间
* partition\*\*（int）–创建新表时的分区数。
* create\_if\_missing（boolean）–未实现。如果表不存在，将始终创建该表。
* error\_if\_exist (boolean) -如果不存在，则返回布尔值。如果已经存在，则不会抛出错误。
* persistent（boolean）–加载表的位置，从持久存储中加载True，从临时存储中加载False。
* in\_version (boolean) -添加版本
* version\_log (string) –要添加的日志

**返回：**

无

**返回类型：**

Table

**Examples：**

**>>> from** **arch.api** **import** session

**>>>** session.save\_data([("one", 1), ("two", 2)], "save\_data", "readme", in\_version=**True**, version\_log="a version")

#### save\_data\_table\_meta(kv, data\_table\_name, …)

**save\_data\_table\_meta(*kv*, *data\_table\_name*, *data\_table\_namespace*)**

**参数：**

* kv（dict）–要保存的图元。v应该由JSON序列化
* data\_table\_name（string）–此数据表的表名
* data\_table\_namespace（string）–此数据表的空间

**返回：**

无

**返回类型：**

Table

**Examples：**

**>>> from** **arch.api** **import** session

**>>>** session.save\_data\_table\_meta({"model\_id": "a\_id", "used\_framework": "fate"}, "meta", "readme")

#### table(\*args, \*\*kwargs)

**table(*\*args*, *\*\*kwargs*) → [arch.api.base.table.Table](https://fate.readthedocs.io/en/latest/doc/api/table_api.html" \l "arch.api.base.table.Table" \o "arch.api.base.table.Table)**

参数：

* name（string）–结果表的表名。
* namespace（string）–结果表的表名称空间。
* partition（int）–创建新表时的分区数。
* create\_if\_missing（boolean）–未实现。如果表不存在，将始终创建该表。
* error\_if\_exist (boolean) -如果不存在，则返回布尔值。如果已经存在，则不会抛出错误。
* persistent（boolean）–加载表的位置，从持久存储中加载True，从临时存储中加载False。
* in\_place\_computing (boolean)–是否启用就地计算。

**返回：**

由加载的数据组成的表

**返回类型：**

Table

**Examples：**

**>>> from** **arch.api** **import** session

**>>>** a = session.table('foo', 'bar', persistent=**True**)

## Table API

### 函数列表

|  |  |
| --- | --- |
| 函数 | 描述 |
| [**collect**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.collect)([min\_chunk\_size, use\_serialize]) | 从表中返回（key，value）2元组的迭代器 |
| [**count**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.count)() | 返回表中的元素数。 |
| [**delete**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.delete)(k[, use\_serialize]) | 返回与键对应的已删除值。 |
| [**destroy**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.destroy)() | 销毁此表，释放其关联的存储资源。 |
| [**filter**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.filter)(func) | 返回一个新表，其中只包含满足通过func传入的函数。 |
| [**first**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.first)([keysOnly, use\_serialize]) | 返回表的第一个元素。 |
| [**flatMap**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.flatMap)(func) | 通过函数func，然后展平它来返回一个新表。 |
| [**get**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.get)(k[, use\_serialize, maybe\_large\_value]) | 获取值匹配键。 |
| [**glom**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.glom)() | 将每个分区中的所有元素合并到一个列表中。 |
| [**join**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.join)(other, func) | 返回一个表，其中包含self和other中具有匹配键的所有元素对，即“inner join”。  每对元素都将作为（k，func（v1，v2））元组返回，其中（k，v1）在self中，（k，v2）在other中。 |
| [**map**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.map)(func) | 通过对该表的每个（键、值）2元组应用函数，返回一个新表。 |
| [**mapPartitions**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.mapPartitions)(func) | 通过将函数应用于该表的每个分区来返回一个新表。 |
| [**mapPartitions2**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.mapPartitions2)(func) | 通过将函数应用于该表的每个分区来返回一个新表。 |
| [**mapValues**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.mapValues)(func) | 通过对该表的每个值应用函数返回一个表，而键不变。 |
| [**put**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.put)(k, v[, use\_serialize, maybe\_large\_value]) | 保存键值记录 |
| [**put\_all**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.put_all)(kv\_list[, use\_serialize, chunk\_size]) | List（key，value）2元组放入表中 |
| [**put\_if\_absent**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.put_if_absent)(k, v[, use\_serialize]) | 仅在未设置键时存储键-值记录。 |
| [**reduce**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.reduce)(func[, key\_func]) | 使用指定的关联二进制运算符减少此表的元素。 |
| [**sample**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.sample)(fraction[, seed]) | 返回此表的抽样子集。 |
| [**save\_as**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.save_as)(name, namespace[, partition, …]) | 将临时表转换为持久表。 |
| [**subtractByKey**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.subtractByKey)(other) | 返回一个只包含这个表中的元素而不包含另一个表中的元素的新表。将临时表转换为持久表。 |
| [**take**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.take)([n, keysOnly, use\_serialize]) | 返回表的前n个元素。 |
| [**union**](https://fate.readthedocs.io/en/latest/doc/api/table_api.html#arch.api.base.table.Table.union)(other[, func]) | 返回这个表和另一个表的并集。 |

#### collect([min\_chunk\_size, use\_serialize])

**abstract collect(*min\_chunk\_size=0*, *use\_serialize=True*) → list**

**参数：**

min\_chunk\_size (int) -如果没有命中表的末尾，返回的最小块大小(关键字节+值字节)。0表示默认块大小(partition\_num \* 1.75 MB)，负数表示没有块大小限制，即返回所有记录。如果用户没有特殊需要，建议使用默认块大小。

**返回：**

无

**返回类型：**

Iterator

**Examples**

**>>>** a = session.parallelize(range(10))

**>>>** b = a.collect(min\_chunk\_size=1000)

>>>list(b)

[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]

#### count()

**参数：**

返回表中元素的数量

**返回：**

表中元素的数量

**返回类型：**

Int

**Examples:**

**>>>** a = session.parallelize(range(10))

**>>>** a.count()

10

#### delete(k[, use\_serialize])

**abstract delete(*k*, *use\_serialize=True*)**

**参数：**

* k (object) – 关键对象。将序列化。必须小于512字节。

**返回：**

已删除键的对应值。如果键不存在，返回None。

**返回类型：**

Object

**Examples:**

**>>>** a = session.parallelize(range(10))

**>>>** a.delete(1)

1

#### destroy()

**参数：**

销毁该表，释放其关联的存储资源。

**返回：**

无

**返回类型：**

None

**Examples:**

**>>>** a = session.parallelize(range(10))

**>>>** a.destroy()

#### filter(func)

**参数：**

* func (k, v -> bool)-返回布尔值的函数。

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize([0, 1, 2])

**>>>** b = a.filter(**lambda** k, v : k % 2 == 0)

**>>>** list(b.collect())

[(0, 0), (2, 2)]

**>>>** c = a.filter(**lambda** k, v : v % 2 != 0)

**>>>** list(c.collect())

[(1, 1)]

#### first([keysOnly, use\_serialize])

**abstract first(*keysOnly=False*, *use\_serialize=True*)**

**参数：**

* keysOnly (boolean) -是否只返回键。True只返回键，False同时返回键和值。
* use\_serialize (boolean) –

**返回：**

表的第一个元素。如果keysOnly=False，它是一个元组;如果keysOnly=True，它是一个对象。

**返回类型：**

tuple or object

**Examples:**

**>>>** a = session.parallelize([1, 2, 3])

**>>>** a.first()

(1, 1)

#### flatMap(func)

**参数：**

* func (k, v -> list) – 应用于每个2元组的函数。

**返回：**

应用函数后，每个列表中包含所有扁平元素的列表。如果密钥存在于多个列表中，则使用“Last-Write Win”策略。

**返回类型：**

List

**Examples:**

**>>> import** **random**

**>>> def** foo(k, v):

result = []

r = random.randint(10000, 99999)

for i in range(0, k):

result.append((k + r + i, v + r + i))

return result

**>>>** a = session.parallelize(range(5))

**>>>** b = a.flatMap(foo)

**>>>** list(b.collect())

[(83030, 83030), (84321, 84321), (84322, 84322), (91266, 91266), (91267, 91267), (91268, 91268), (91269, 91269), (71349, 71349), (71350, 71350), (71351, 71351)]

#### get(k[, use\_serialize, maybe\_large\_value])

**abstract get(*k*, *use\_serialize=True*, *maybe\_large\_value=False*)**

**参数：**

k (key object) – 将序列化，key大小必须小于512字节。

**返回：**

键的对应值。如果键不存在，返回None。

**返回类型：**

Object

**Examples:**

**>>>** a = session.parallelize(range(10))

**>>>** a.get(1)

(1, 1)

#### glom()

**参数：**

无

**返回：**

包含所有合并分区及其元素的列表。每个元组的第一个元素从每个分区的最后一个元素的键中选择。

**返回类型：**

List

**Examples:**

**>>>** a = session.parallelize(range(5), partition=3).glom().collect()

**>>>** list(a)

[(2, [(2, 2)]), (3, [(0, 0), (3, 3)]), (4, [(1, 1), (4, 4)])]

#### join(other, func)

**参数：**

* other (Table) –另一个要操作的表
* func (v1, v2 -> v) – 二元运算符，应用于键存在于两个表中的值

**返回：**

包含结果的表。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize([('a', 1), ('b', 4)], include\_key=**True**)

**>>>** b = session.parallelize([('a', 2), ('c', 3)], include\_key=**True**)

**>>>** c = a.join(b, **lambda** v1, v2: v1 + v2)

**>>>** list(c.collect())

[('a', 3)]

#### map(func)

**参数：**

* func (k1, v1 -> k2, v2) – 应用于每个二元组的函数。

**返回：**

包含结果的新表。。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize(['a', 'b', 'c']) *# [(0, 'a'), (1, 'b'), (2, 'c')]*

**>>>** b = a.map(**lambda** k, v: (v, v + '1'))

**>>>** list(b.collect())

[("a", "a1"), ("b", "b1"), ("c", "c1")]

#### mapPartitions(func)

**参数：**

* func (iter -> v) –应用于每个分区的函数。

**返回：**

一个具有k-v: uuid键-v的新表。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize([1, 2, 3, 4, 5], partition=2)

**>>> def** f(iterator):

**>>>**  sum = 0

**>>>**  **for** k, v **in** iterator:

**>>>**  sum += v

**>>>**  **return** sum

**>>>** b = a.mapPartitions(f)

**>>>** list(b.collect())

[(3, 6), (4, 9)]

#### mapPartitions2(func)

**参数：**

* func (iter -> (k, v)) –应用于每个分区的函数。

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>> from** **arch.api** **import** session

**>>>** a = session.parallelize([1, 2, 3, 4, 5], partition=2)

**>>> def** f(iterator):

**...**  s = 0

**...**  **for** k, v **in** iterator:

**...**  s += v

**...**  **return** [(s, s)]

**...**

**>>>** b = a.mapPartitions2(f)

**>>>** list(b.collect())

[(6, 6), (9, 9)]

#### mapValues(func)

**参数：**

* func (v1 -> v2) – The function applying to each value.

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>> from** **arch.api** **import** session

**>>>** a = session.parallelize([('a', ['apple', 'banana', 'lemon']), ('b', ['grapes'])], include\_key=**True**)

**>>>** b = a.mapValues(**lambda** x: len(x))

**>>>** list(b.collect())

[('a', 3), ('b', 1)]

#### put(k, v[, use\_serialize, maybe\_large\_value])

**abstract put(*k*, *v*, *use\_serialize=True*, *maybe\_large\_value=False*)**

**参数：**

* k (Key object) – 将被序列化。必须小于512字节
* v (object) – 将被序列化。必须小于32mb (或eggroll 2.x 小于2G, 依赖实现)
* use\_serialize (bool, defaults True) –默认为True
* maybe\_large\_value (bool, defaults False) –默认为True

**Examples:**

**>>> from** **arch.api** **import** session

**>>>** a = session.parallelize(range(10))

**>>>** a.put('hello', 'world')

**>>>** b = a.collect()

**>>>** list(b)

[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9), ('hello', 'world')]

#### put\_all(kv\_list[, use\_serialize, chunk\_size])

**abstract put\_all(*kv\_list:* *Iterable*, *use\_serialize=True*, *chunk\_size=100000*)**

从可迭代项中放入(键，值)二元流。

元素必须是精确的二元组，它们可能不是任何其他类型，或元组的子类。

**参数：**

* kv\_list (Iterable) – 键-值二元数组iterable。将序列化。每个键必须小于512字节，值必须小于32 MB(取决于实现)。

**Examples:**

**>>>** a = session.table('foo', 'bar')

**>>>** t = [(1, 2), (3, 4), (5, 6)]

**>>>** a.put\_all(t)

**>>>** list(a.collect())

[(1, 2), (3, 4), (5, 6)]

#### put\_if\_absent(k, v[, use\_serialize])

**abstract put\_if\_absent(*k*, *v*, *use\_serialize=True*)**

仅在未设置键时存储键-值记录。

**参数：**

* k (key object) – 将被序列化。必须小于512字节。
* v (Value object) – 将被序列化。必须小于32mb (或eggroll 2.x 小于2G, 依赖实现)

**Examples:**

**>>>** a = sessiojn.parallelize(range(10))

**>>>** a.put\_if\_absent(1, 2)

**>>>** b = a.collect()

**>>>** list(b)

[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]

**>>>** a.put\_if\_absent(-1, -1)

**>>>** list(b)

#### reduce(func[, key\_func])

**abstract reduce(*func*, *key\_func=None*)**

**参数：**

* func (v1, v2 -> v) – 应用于每个2元组的二进制运算符。
* key\_func (k -> k') – 应用于每个键的一元运算符，以获得要还原的实键。默认值为None，这意味着减少原始键。

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>> from** **operator** **import** add

**>>>** session.parallelize([1, 2, 3, 4, 5]).reduce(add)

**>>>** 15

#### sample(fraction[, seed])

**abstract sample(*fraction*, *seed=None*)**

**参数：**

* fraction (float) – 样本的期望大小是这个表的大小的一部分，不需要替换:每个元素被选择的概率。分数必须是[0,1]并进行替换:选择每个元素的预期次数。
* seed (int) – 随机数生成器的种子。当没有传递时，使用当前时间戳。

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>>** x = session.parallelize(range(100), partition=4)

**>>>**  6 <= x.sample(0.1, 81).count() <= 14

True

#### save\_as(name, namespace[, partition, …])

**abstract save\_as(*name*, *namespace*, *partition=None*, *use\_serialize=True*, *\*\*kwargs*)**

**参数：**

* name (string) – 结果表的表名。
* namespace (string) – 结果表的表名称空间。
* partition (int) – 新持久表的分区数。
* use\_serialize –

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize(range(10))

**>>>** b = a.save\_as('foo', 'bar', partition=2)

#### subtractByKey(other)

返回一个只包含这个表中的元素而不包含另一个表中的元素的新表。

**参数：**

* **other (Table) – 另一个要操作的表。**

**返回：**

包含结果的新表。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize(range(10))

**>>>** b = session.parallelize(range(5))

**>>>** c = a.subtractByKey(b)

**>>>** list(c.collect())

[(5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]

#### take([n, keysOnly, use\_serialize])

**abstract take(*n=1*, *keysOnly=False*, *use\_serialize=True*)**

**参数：**

* n (int) – 返回的顶部数据数。
* keysOnly (bool) – 是否只返回键。True只返回键，False同时返回键和值。

**返回：**

前n个键或键-值对的列表。

**返回类型：**

List

**Examples:**

**>>>** a = session.parallelize([1, 2, 3])

**>>>** a.take(2)

[(1, 1), (2, 2)]

**>>>** a.take(2, keysOnly=**True**)

[1, 2]

#### union(other[, func])

**abstract union(other, func=<function Table.<lambda>>)**

如果启用，就地计算将应用。结果将在左表中(调用者)。

**参数：**

* other (Table) – 另一个要操作的表。
* func (v1, v2 -> v) – 应用于键同时存在于两个表中的值的函数。默认使用左表的值。

**返回：**

包含结果的表。

**返回类型：**

Table

**Examples:**

**>>>** a = session.parallelize([1, 2, 3]) *# [(0, 1), (1, 2), (2, 3)]*

**>>>** b = session.parallelize([(1, 1), (2, 2), (3, 3)])

**>>>** c = a.union(b, **lambda** v1, v2 : v1 + v2)

**>>>** list(c.collect())

[(0, 1), (1, 3), (2, 5), (3, 3)]

# 5. EggRoll外部的输入和调用入口

引入的python脚本

<https://github.com/FederatedAI/FATE/blob/master/arch/api/session.py>

外部输入csv/生成数据

# 6. 测试脚本函数（集群）

from arch.api import session, WorkMode, Backend

from arch.api.utils import log\_utils, file\_utils, dtable\_utils, version\_control

import random

session.init("a\_job\_id", WorkMode.CLUSTER, Backend.EGGROLL)

b=session.save\_data([("one", 1), ("two", 2)], "save\_data", "readme", in\_version=True, version\_log="a version")

print(b.count())

print(b.first())

c = b.map(lambda k, v: (k, v+1))

print(list(c.collect()))

d = b.join(c, lambda v1, v2: v1 + v2)

print(list(d.collect()))

print(d.reduce(lambda v1,v2:v1+v2))

e=d.filter(lambda k,v:k=='two')

print(list(e.collect()))

def foo(k, v):

result = []

r = random.randint(10000, 99999)

for i in range(0, 3):

result.append((k, v + r + i))

return result

f=d.flatMap(foo)

print(list(f.collect()))

print(f.get("one"))

g=f.glom().collect()

print(list(g))

def foo1(iterator):

s = 0

for k, v in iterator:

s += v

return [(k, s)]

a1=f.mapPartitions2(foo1)

print(list(a1.collect()))

a2=session.save\_data([('a', ['apple', 'banana', 'lemon']), ('b', ['grapes'])], "save\_data1", "readme1", in\_version=True, version\_log="a version1")

a3=a2.mapValues(lambda x: len(x))

print(list(a2.collect()))

print(list(a3.collect()))

c.put("hello", 5)

print(list(c.collect()))

a5 = [("h1", 6), ("h2", 7), ("h3", 8)]

c.put\_all(a5)

print(list(c.collect()))

b1=c.subtractByKey(b)

print(list(b1.collect()))

print(b1.take(2))

print(b1.take(3, keysOnly=True))

print(list(b.collect()))

print(list(c.collect()))

b2=b.union(c,lambda v1,v2:v1+v2)

print(list(b2.collect()))