An In-depth Journey into Odoo's ORM

Pham Thi Ngoc Mai

Onnet - AHT

September 17th, 2021

Introduction

Implementation requirements

- Be correct
- Be secure:
  - access rights
  - against external attacks, sql injections
- Be effient:
  - scalable algorithm
  - few and efficient SQL queries: the cost in term of time of every SQL query is huge compared to the cost of simple computation in python code

Key data structures

- Registry
- Record cache
- Fields to write
- Fields to compute
- Field triggers

Registry

What?

A place where every model name is associated to a python class

```
class Registry(Mapping):
    """ Model registry for a particular database.

    The registry is essentially a mapping between
        model names and model classes.
    There is one registry instance per database.

    """
    ...
```

```python
# Model definitions
class Foo0(models.Mode):
    _name = 'foo'
    ...
class Foo1(models.Model):
    _inherit = 'foo'
    ...
class Foo2(models.Model):
    _inherit = 'foo'
    ...
```

```python
# Model classes
class foo(Foo2, Foo1, Foo0, Base):
    _name = 'foo'
    ...
```

- Goal: map **model_name** to **model_class**
- **model_class** should reflect model definitions
- **browse()** returns an instance of **model_class**
- holds metadata

Why?

- Determine fields by introspection (which are fields defined by model?)
- Add custom fields
- Add `_inherits` fields
- Setup fields (parameters, depends, ...)

Challenges:

- Getting the **model classes** right
- **Performance** of setup

Record cache

Cache

Caching is one of the ways to achieve fast and responsive applications

- an area of memory which is of high speed
- it stores often used data
- to serve up data faster than is possible by accessing the data's primary storage location.

Caching strategies

```
self.envs = WeakSet()
```

- algorithms to define which data should stay in memory and which should be removed

Strategies	Eviction policy	Use case
FIFO	Evicts the oldest of the entries	Newer entries are most likely to be reused
LIFO	Evicts the latest of the entries	Older entries are most likely to be reused
LRU	**Evicts the least recently used entry**	**Recently used entries are most likely to be reused**
MRU	Evicts the most recently used entry	Least recently used entries are most likely to be reused
LFU	Evicts the least often accessed entry	Entries with a lot of hits are more likely to be reused

LRU strategy

- Every time you access an entry, the LRU algorithm will move it to the top of the cache.
- Identify the entry that's gone unused the longest by looking at the bottom of the list.
- O(1)

Record cache

- Goal: **augmented** database cache
- Stores **field values**
- Accessible on **environment**

Record cache

api.py

```
class Cache(object):
    """ Implementation of the
    cache of records. """
    def __init__(self):
        self._data = defaultdict(dict)
...
```

```
class Environments(object):
    """ A common object for all
    environments in a request. """
    def __init__(self):
        # cache for all records
        self.cache = Cache()
...
```

Record cache prefetching

```
records = model.browse(ids)
for record in records:
    print(record.name)
```

Record cache prefetching

- First iteration:
  - look up cache → nothing
  - fetch field `record`:
    - look up record._prefetch_ids for missing value → all `ids`
    - if field is a column, fetch all columns
    - `records._read(fields)`
  - look up cache → value
- Next iterations:
  - look up cache → value

Record cache data structure

```
{field: {record_id: value}}
```

- Access N records → 1 + N dict lookups
- Update N records → 1 + N dict updates
- Invalidate N records → 1 + N dict deletions

Record cache data structure

```
{field: {ctx_key: {record_id: value}}}
```

- Access N records → 2 + N dict lookups
- Update N records → 2 + N dict lookups
- Invalidate N records → 1 + K*N dict lookups

Challenges

- cache consistency
- invalidate
- update

try to avoid invalidaing the cache as much as possible