

Conclusion

▼ Conclusion

▼ S1. 项目基础

- S1.1. 项目架构
- S1.2. 装饰器
- S1.3. 魔法方法
- S1.4. 使用 `os` 包
- S1.5. 使用 `secrets` 包
- S1.6. 使用 `typing` 包
- S1.7. 使用 `werkzeug` (tool) 包

▼ S2. Jinja2 模板

▼ S2.1. 语法块

- S2.1.1. 条件 `if`
- S2.1.2. 循环 `for`

▼ S2.2. 模板继承组合

- S2.2.1. 块继承 `extends` block
- S2.2.2. 包括 `include`

▪ S2.3. 过滤器

▪ S2.4. 测试

▼ S2.5. 模板约定

- S2.5.1. 基础模板
- S2.5.2. 子模板
- S2.5.3. 功能模板

▼ S3. flask API

▪ S3.1. Flask 应用对象

▪ S3.2. Blueprint 蓝图对象

▼ S3.3. 全局[应用/请求]上下文 context

- S3.3.0. `@app.context_processor`
- S3.3.1. `g`
- S3.3.2. `current_app`
- S3.3.3. `request`

▪ S3.4. 装饰器 wrapper

▼ S3.5. 实用函数方法

- S3.5.1. 路由方法
- S3.5.2. 前后数据传输

- S3.6. request 全局变量
- ▼ S4. flask-SQLAlchemy, migrate: 数据库与数据模式及迁移
 - S4.1. 初始化数据库和迁移脚本
 - ▼ S4.2. 数据迁移命令
 - S4.2.0. 命令选项说明
 - S4.2.1. 查看版本
 - S4.2.2. 提交版本
 - S4.2.3. 修改当前版本
 - ▼ S4.3. 数据模型建立
 - S4.3.1. 数据类型 Type
 - S4.3.2. 映射器配置 mapper
 - S4.3.3. 映射注释方法 mapper
 - S4.3.4. 映射类继承 Mixins
 - ▼ S4.4. 关系模型
 - S4.4.1. 一对多 ORM
 - S4.4.2. 一对一 ORM
 - S4.4.3. 多对多 ORM
 - S4.4.4. 关联对象
 - ▼ S4.5. 映射类查询
 - S4.5.0. Result 查询结果对象
 - S4.5.1. Select 选择对象
 - S4.5.2. where 条件子句
 - S4.5.3. FROM 子句和 Joins 连接方式
 - S4.5.4. 别名 alias 和标签 label
 - S4.5.5. 子查询和 CTE
 - S4.5.6. SQL 函数
 - ▼ S4.6. 映射类行为
 - S4.6.1. 插入 INSERT
 - S4.6.2. 更新 UPDATE
 - S4.6.3. 删除 DELETE
 - S4.7. 自定义 SQL 语句
- ▼ S5. flask-WTF: 表单
 - S5.1. 创建表单
- S6. flask-Login: 用户认证
- Sx. 网页路由设计
- Sy. 测试

S1. 项目基础

S1.1. 项目架构

先要配置虚拟环境，可以通过 `venv` 创建，进入虚拟环境内再在虚拟环境内安装 `requirements` 下载对应的 `python` 包，其中含有 `flask`, `dotenv` 等.

构建以下项目结构，其中 `.venv` 通过 `python -m venv .venv` 自动生成：

```
microblog/
| .flaskenv
| .env # .ignore
| app.sqlite # .ignore <auto-generated>
| config.cfg # .py, .json, .toml, ...
| requirements.txt
| wsgi.py
|
├─.venv/ # .ignore <auto-generated>
|
├─migration/ # .ignore <auto-generated>
|
└─apps/
    | __init__.py
    | views.py
    | forms.py
    | models.py
    |
    └─templates/
        | base.html
        └─ index.html
```

- `.flaskenv` 写入应用开发环境的配置信息，如 `FLASK_APP=wsgi.py` , `FLASK_DEBUG=1` , 无需编程配置;
- `.env` 写入应用的配置私密信息，如 `MAIL_USERNAME` , `MAIL_PASSWORD` 等，需要通过 `config.cfg` 编程配置到应用;
- `base.html` 是基础子模版，里面含有首页、用户页、设置页等公共内容（选项、登入注册、登出、个人、导航栏）
- `index.html` 是 SEO 默认搜索指定的 HTML 文件名，通常用作根地址 `URL /` 的页面（首页），游客可访问.

- 推荐使用 `config.py` 因为可以配置在找不到系统变量下的默认值. 其中对于私密信息的配置使用 `<INFO_NAME> = os.environ.get('<INFO_NAME_IN_ENV>') or <default_value>` 的方式配置.

```
# microblog/config.py
class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'your-secret-key'
    ...
```

S1.2. 装饰器

python 内 `self` 代表**实例**, `cls` 代表**类本身**.

python 内的装饰器就如同函数的函数, 能够为函数提供动态修改, 增加属性.

- `@property` 能够让类的方法像属性一样调用, 并能够设置 `getter` 和 `setter` 方法.

```
class Student(object):
    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value

    @property
    def gpa(self):
        return (score - 60) / 10

s = Student()
s.score = 80 # OK, 实际转化为 s.set_score(80)
s.score # 显示 60. 实际转化为 s.get_score()
s.score = 9999 # 出错 ValueError. 会进入 s.set_score(9999) 由于大于 100 报错
s.gpa # 显示 2.0
s.gpa = 3.9 # 出错 AttributeError. 因为 gpa 是只读属性, 只能通过 score 设置.
```

- `@classmethod` 使得类内的方法可以不通过实例化调用.

```

class A:
    bar = 1

    def foo(self):
        print(self.bar)

    # @classmethod will pass cls as an argument
    # when use A.class_foo() -> class_foo(A.cls)
    # without would raise error
    # when use A.class_foo() -> class_foo() since no argument passed
    @classmethod
    def class_foo(cls):
        print(cls.bar)

    # informal usage but runnable
    def non_foo():
        print(A.bar)

a = A()
a.bar = 2
a.foo() # 2. foo(a.self)
a.class_foo() # 1. class_foo(a.cls)
a.non_foo() # Error. because a.func() -> func(a.self) but non_foo take 0 argument
A.foo() # Error. foo(A.self) but A is not an instance
A.class_foo() # 1. class_foo(A.cls)
A.non_foo() # 1. non_foo()

# so use a.foo() and A.class_foo()

```

S1.3. 魔法方法

- `def __repr__(self)` : 通过终端直接查看类对象的信息或是 `print` 打印显示的信息
- `def __str__(self)` : 通过 `print` 打印显示的信息, 会覆盖 `__repr__` 的内容

S1.4. 使用 `os` 包

- `path` : 处理文件路径
 - `dirname(__file__)` 获取当前所在文件的父目录的绝对路径
 - `abspath(__file__)` 获取当前所在文件的绝对路径
 - `join(*str)` 拼接路径, 从左到右找第一个[有 `\` 开头的]绝对路径, 往后拼接.
- `environ` : 处理环境变量

- `get('<ENV_NAME>')` 获取环境变量, 若不存在则返回 `None`

S1.5. 使用 `secrets` 包

- `token_hex(nbyte: int | None = None) -> str` 生成 `n` 字节长的随机字符串, 通常用于生成密钥.

S1.6. 使用 `typing` 包

即 `type hint` 类型提示和规定.

- `Option[Type]` 即 `Type | None` 类型

S1.7. 使用 `werkzeug` (tool) 包

- `werkzeug.security`: 安全模块
 - `generate_password_hash(password)`: 生成密码的哈希值
 - `check_password_hash(hash, password)`: 检查密码的哈希值是否匹配

S2. Jinja2 模板

S2.1. 语法块

S2.1.1. 条件 `if`

```
{% if condition %}  
...  
{% elif condition %}  
...  
{% else %}  
...  
{% endif %}
```

S2.1.2. 循环 `for`

```
{% for item in items [if condition] %}  
... {# 循环内可以使用特殊变量 loop 帮助器 #}  
{% else %}  
... {# 迭代体空时 #}  
{% endfor %}
```

S2.2. 模板继承组合

S2.2.1. 块继承 extends block

```
{# base.html #}
{% block name %}Base{% endblock [name] %}<br>
{% block name1 %}Base1{% endblock %}<br>
{% block name2 required%}Base2{% endblock %}<br>
{% block name3 %}Base3{% endblock %}

{# index.html #}
{% extends "base.html" %} {# 会把 base.html 内的内容都继承，只有 block 部分可以改变 #}
{% block name %}Index{% endblock [name] %}
{% block name3 %}Index3<br>super()<br>self.name1(){% endblock %}
Index Itself {# 子模板内的其它内容都不会显示 #}

{# 最终显示 #}
Index
Base1
TemplateRuntimeError {# 并不是这个信息，而是整个页面出错 #}
Index3
Base3
Base1
```

S2.2.2. 包括 include

```
{% include "lib1.html" %}
{% include "lib2.html" without context %}
{% include "lib3.html" ignore missing with context %}
{# ignore missing 必须在 with[out] context 之前 #}
{% include ["lib4.html", "lib5.html"] ignore missing %}
```

S2.3. 过滤器

S2.4. 测试

S2.5. 模板约定

S2.5.1. 基础模板

用 base.html 命名：

```

<!DOCTYPE base.html>
<html lang="zh">
<head>
  <meta charset="UTF-8">
  <meta name="description" content="{# TODO: 描述应用页面 #}">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="icon" href="{# TODO: 放置浏览器页面图标 #}"/>
  <title>
    {% if title %}
    {{ title }} - {# TODO: 应用系统的名字, 如 StackOverflow #}
    {% else %}
    {# TODO: 应用系统的名字 - 应用系统的描述 #}
    {% endif %}
  </title>
</head>
<body>
  {% block head/navigation %}
  {# 导航栏 #}
  {% endblock %}
  {% block content %}
  {# 首页主要内容 #}
  {% endblock %}
  {% block footer %}
  {# 页尾状态栏 #}
  {% endblock %}
</body>
</html>

```

S2.5.2. 子模板

用 index.html 等具体功能名字命名:


```
{% extends "base.html" %}

{% block head/navigation %}
{# 导航栏 #}
{% endblock %}
{% block content %}
{# 首页主内容 #}
{% endblock %}
{% block footer %}
{# 页尾状态栏 #}
{% endblock %}
```

S2.5.3. 功能模板

用 `_post.html` 命名, 开头用 `_` 表示, 以具体内容为名字:

```
<table>
  <tr valign="top">
    <td></td>
    <td>
      <a href="{{ url_for('user', username=post.author.username) }}">{{ post.author.username }}·
      says: <br>
      {{ post.body }}
    </td>
  </tr>
</table>
```

S3. flask API

S3.1. Flask 应用对象

是应用本身, 需要通过 `app = Flask(__name__)` 创建最基本的应用 `app`, 在运行时会有对应的 `current_app` 可以从 **application context 应用上下文** 中找到运行中的应用

S3.2. Blueprint 蓝图对象

S3.3. 全局[应用/请求]上下文 context

S3.3.0. @app.context_processor

flask 提供了一类装饰器，用于在上下文中添加全局变量，例如

```
@app.shell_context_processor
def make_shell_context():
    return dict(**kwargs)
```

这使得 flask shell 上下文会自动加入 `**kwargs` 数据.

S3.3.1. g

`g` 是一个全局变量，是应用任意时刻都能使用的变量，在每个请求中也会不同.

S3.3.2. current_app

application context 应用上下文，对于无需导入应用程序或无法导入应用程序的地方，可以通过 `current_app` 获取当前运行的 Flask 实例 `app`，以 **proxy 代理** 的形式获取当前应用的信息，但如果要进行底层操作，需要通过 `current_app._get_current_object` 获取 `app` 对象.

S3.3.3. request

request context 请求上下文，这个功能较多，具体内容见 [request 全局变量](#)

S3.4. 装饰器 wrapper

S3.5. 实用函数方法

S3.5.1. 路由方法

- `flask.redirect()`：重定向到某一 URL.

```
def redirect(
    location: str, # '<url>' 路径
    code: int = 302,
    Response: type[Response] | None = None
) -> Response
```

- flask.url_for() : 生成 URL.

```
def url_for(
    endpoint: str, # '<route_name>'. 即由 `@app.route()` 包装的函数名
    *,
    _anchor: str | None = None, # 给定 URL 锚点, 即 `/url#anchor`
    _method: str | None = None, # 给点 method.
    _scheme: str | None = None,
    _external: bool | None = None,
    # 可以再加入 key=value 的参数, 作为路由器的参数或查询 `/url?key1=value1&key2=value2...`
    **values
) -> str

@app.route("/index/<id>")
def index(id):
    pass

url_for("index", _anchor="anchor", id=2, key=value)
# '/index/2#anchor?key=value'
```

S3.5.2. 前后数据传输

- flask.flash() / get_flashed_messages() : 前者方法向下一个请求闪现一条消息, 模板通过后者方法获取.

```
def flash(
    message: str, # 要闪现的消息
    category: str # 消息类别
    # 建议使用 'message', 'error', `info`, 'warning'
)

def get_flashed_messages(
    with_categories: bool = False, # 是否返回类别, 元组或单纯信息
    category_filter: Iterable[str] = () # 根据消息类别过滤
) -> list[str] | list[tuple[str, str]]
```

S3.6. request 全局变量

request 记录了每次请求的信息, 直接通过 request.api 使用可以得到对应的作用.

S4. flask-SQLAlchemy, migrate: 数据库与数据模式及迁移

S4.1. 初始化数据库和迁移脚本

在 `microblog/apps/__init__.py` 中初始化数据库和迁移脚本对象.

```
# microblog/apps/__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
```

在 `microblog` 路径下输入以下终端命令:

```
(.venv) microblog> flask db init
# 会通过 .flaskenv 设置的 FLASK_APP 位置寻找 migrate 对象
```

由此会创建 `microblog/migrations` 文件夹, 其中包含迁移脚本.

在目录 `microblog/migrations/versions/` 下可以查看具体的脚本, 以 `<version_hash>_<commit_message>.py` 命名.

S4.2. 数据迁移命令

S4.2.0. 命令选项说明

- `--help`: **都有**, 帮助手册
- `-d, --directory <dir_name>`: **都有**, 指定迁移脚本目录, 默认是 `"migrations"`.
- `-v, --verbose`: 详细信息输出.
- `-m, --message "<commit_message>"`: 提交的版本信息.
- `--sql`: 不执行命令, 只输出 SQL 语句.

S4.2.1. 查看版本

- `flask db current`: 查看当前数据库版本.
 - `-v, --verbose`

- flask db history : 查看所有迁移脚本版本.
 - -v, --verbose
- flask db check : 查看是否有新版本.

S4.2.2. 提交版本

修改 models 代码后提交, 如同 git commit .

- flask db migrate : 将当前数据库模式提交到迁移脚本中.
 - -m, --message "<commit_message>"
 - --sql

S4.2.3. 修改当前版本

- flask db upgrade [<version_hash>=HEAD] : 将数据库模式回滚到指定版本. 不给出版本时**默认最新版本** HEAD ;
 - --sql
- flask db downgrade [<version_hash>=-1] : 将数据库模式升级到指定版本. 不给出版本时**默认降一级**, 可以使用 BASE 降级到最初空数据库版本;
 - --sql

S4.3. 数据模型建立

在 models 内建立数据模型, 代码格式如下:

```

import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

# Imperative/Classic Mapping 命令/经典式映射
table_name = Table(
    "table_name",
    db.metadata,
    Column("attr1", sa.TypeEngine(*args), **kwargs),
    ...
)

# Declarative Mapping 声明式映射
class TableName(db.Model):
    __tablename__ = "table_name" # 可省略, 省略自动生成类同名表
    attr1: so.MappedMethod[Type] = sa.mapped_column(sa.TypeEngine(*args), **kwargs)
    ...
    rel_table1: so.MappedMethod['TableName'] = so.relationship(**kwargs)

    def func():
        # 这里封装该表的常用数据处理和增删改查操作
        pass

# 通过 print(sqlalchemy.schema.CreateTable(TableName.__table__))
# 可以查看创建表 TableName 的创建 SQL 语句、

```

S4.3.1. 数据类型 Type

可以根据 Declarative 声明式映射表中的 annotation 类型注释给出, 即 `attr: so.Mapped[Type]` 这一部分. 通常通过这种方法设定的数据类型不用再在 `sa.mapped_column()` 中给出, 但是由于 `Type` 无法指定字符串长度之类的选项, 所以这种情况还是需要从 `sa.mapped_column()` 内给出.

```

from sqlalchemy import types

# 默认的类型映射
# 通过 Mapped 将 Type 映射到 TypeEngine 的类型
type_map: Dict[Type[Any], TypeEngine[Any]] = {
    bool: types.Boolean(),
    bytes: types.LargeBinary(),
    datetime.date: types.Date(),
    datetime.datetime: types.DateTime(),
    datetime.time: types.Time(),
    datetime.timedelta: types.Interval(),
    decimal.Decimal: types.Numeric(),
    float: types.Float(),
    int: types.Integer(),
    str: types.String(),
    uuid.UUID: types.Uuid(),
    enum.ENUM: types.Enum(),
    typing.Literal: types.Enum(),
}

```

- 可为空的数据类型:

- 可以通过 `sa.mapped_column(nullable=True)` 的 `nullable` 参数设定, 默认为 `False`.
- 也可以直接通过 `from typing import Optional` 来设定, 如

```
attr: so.Mapped[Optional[Type]] .
```

两种设定违背也可运行, 但数据库内遵从 `nullable` 设定, python 内遵从 `Optional` 设定.

- 对于枚举类型, 如上 `type_map` 给出了两种方法, 一种是通过 `class MyEnum(enum.ENUM)` 定义并通过 `attr: so.Mapped[MyEnum]` 映射, 另一种是通过

```
attr: so.Mapped[typing.Literal["enum1", "enum2", ...]] 映射.
```

但是对于不同的数据库 SQL 语言, 通过 `ENUM` 和 `Literal` 所映射的不一定是 `SQL.Enum`, 有可能是 `VARCHAR`, 例如 PostgreSQL 使用的 `native_enum` 是 `TYPE`, 通过 `ENUM` 会创建, 但是使用 `Literal` 则不会, 所以建议**就使用 `ENUM` 创建**.

- 对于重复定义的类型如 `id`, `name` 等, 可以使用 python 内置库 `typing` 中的 `Annotated` 定义一种数据类型, 使用方法如下:

```

from typing import Annotated
import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

id_pk = Annotated[int, so.mapped_column(primary_key=True)]

class User(db.Model):
    id: so.Mapped[id_pk]
    # equal to `id: so.Mapped[int] = so.mapped_column(primary_key=True)`

```

- 对于复合类型, 使用 `composite` 定义:

```

import dataclasses
from sqlalchemy.orm import DeclarativeBase, Mapped
from sqlalchemy.orm import composite, mapped_column

@dataclasses.dataclass
class Point:
    x: int
    y: int

class Vertex(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True)

    start: Mapped[Point] = composite(mapped_column("x1"), mapped_column("y1"))
    end: Mapped[Point] = composite(mapped_column("x2"), mapped_column("y2"))

```

S4.3.2. 映射器配置 mapper

1. `mapped_column`: 是基本的创建列的方法. 所有参数都是可选的, 如果不传入参数, 甚至可以只用 `attr: so.Mapped[Type]` 来声明.
 - `__name: str`: 表名, 作为第一个参数传入;
 - `__type: TypeEngine`: 如果有参数 `__name` 则作为第二个参数传入, 否则作为第一个参数传入;
 - `*args`: 附加的位置参数包括构造 `ForeignKey`, `CheckConstraint`, `Identity`, 用于约束限制;
 - `primary_key: bool`: 默认 `False`;
 - `nullable: bool`: 主键默认 `False`, 其他默认 `True`;
 - `unique: bool`: 默认 `False`;
 - `index: bool`: 默认 `False`;
 - `default: Type`: 默认值. (注意如果输入的值是动态变化的, 如 `datetime`, 需要使用 `lambda` 函数作为输入值, SQLAlchemy 会在创建元组时动态调用该函数, 否则会始终等于应用开始运

行时的值)

- onupdate : 更新时默认值.
- defer: bool: 默认 False , 延迟加载, 常用于不频繁访问的大数据;
- active_history: bool: 默认 False , 访问时加载更改前的上一个值, 也适用于通过 get_history() 获取;

2. column_property : 为了能够调用某些计算方法, 如同聚类函数一般, 又不想真实存储在表格内, 可以使用 column_property 来创建一个虚拟属性, 不会存储在表格内, 是 **read only** 的. 见下例

```
class User(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True)
    first_name: Mapped[str] = mapped_column(String(50))
    last_name: Mapped[str] = mapped_column(String(50))
    fullname: Mapped[str] = column_property(first_name + " " + last_name)

print(select(User))
# SELECT
#   "user".first_name || :param_1 || "user".last_name AS anon_1,
#   "user".id,
#   "user".first_name,
#   "user".last_name
# FROM "user"
```

3. deferred : 融合了 column_property 和 mapped_column(deferred=True) 的功能, 能够动态生成懒加载的属性, 并且也没有在数据库内真实定义列属性.

```
class User(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True)
    first_name: Mapped[str] = mapped_column(String(50))
    last_name: Mapped[str] = mapped_column(String(50))
    fullname: Mapped[str] = deferred(first_name + " " + last_name)

print(select(User))
# SELECT
#   "user".id,
#   "user".first_name,
#   "user".last_name
# FROM "user"
```

4. relationship : 关系模式详见 [关系模式](#).

5. @declared_attr : 若需要动态根据表名或表结构配置表, 可以用 @declared_attr 装饰器声明, 类似 python 的 property 装饰器. 通常是用于表继承里, 对于 Mixin 表中的 deferred , relationship , column_property , composite 等都需要使用 @declared_attr 装饰器.

6. `hybrid_property` : 需要对输入的属性进行验证时可以使用 python 的 `property` 装饰器, 但该装饰器无法使用 `select` 方法, 所以 SQLAlchemy 包装了新的 getter/setter 装饰器.

```
from sqlalchemy.ext.hybrid import hybrid_property

class EmailAddress(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True)

    _email: Mapped[str] = mapped_column("email")

    @hybrid_property
    def email(self):
        return self._email[:-12]

    @email.setter
    def email(self, email):
        self._email = email + "@example.com"

    @email.expression
    def email(cls):
        """
        使用 select 时 python 转换成 SQL 语句可能无法正确转换
        通过 hybrid_property_name.expression 定义在 SQL 内的表达式
        """

        return func.substr(cls._email, 0, func.length(cls._email) - 12)
```

S4.3.3. 映射注释方法 mapper

1. `Mapped` : 用于声明映射属性的类型和关系.
2. `WriteOnlyMapped` : 对于相关联集合非常大时, 直接查询随带占用的内存相当大, `WriteOnlyMapped` 来声明只写关系, 会将关系定义为 `relationship(lazy="write_only")`, 查询父表时都会进行懒加载. 这种定义会将对象包装成 **WriteOnlyCollection 对象**, 这使得无需使用查询就可以进行写入 (`insert / update / delete`, `add / add_all / remove`) 操作, 这对于不常读取但写入频繁的属性的性能提高很多, 但无法直接像列表一样直接访问每个项, 不过提供了 `select()` 方法返回查询语句.

```

user = db.session.scalars(sa.select(User)).one()
user.posts[0].body = 'new text'
# Mapped: [<Post 1>, <Post 2>]
# 会进行查询，查询后再更改
user.posts.update(p)
# WriteOnlyMapped: <sqlalchemy.orm.writeonly.WriteOnlyCollection object>
# 不会全部读取，只进行单个修改
posts = db.session.scalars(user.posts.select()).all()
# 此时才会查询全部信息

```



WriteOnlyCollection

posts 会封装 user.id，内部也有方法

- select(): 创建 query 语句 (Select 对象)，再执行，效果同直接映射后的 user.posts .
- update(): 创建 Update 对象.
- delete(): 创建 Delete 对象.
- add(item: _T): 添加单个项.
- add_all(iterator: Iterable[_T]): 批量添加项目.
- remove(item): 删除某个项.

3. DynamicMapped: 动态加载器，会将关系定义为 relationship(lazy="dynamic")，县北部已为遗留选择，应该使用 WriteOnlyMapped .

S4.3.4. 映射类继承 Mixins

Mixins 的思想就是把通用的属性放在一个类中，然后其他类继承这个类，从而减少重复代码，类似于基类继承. 例如每一个实体表都会有 id , create_time , update_time 这三个字段，就可以把这三个字段放在一个类中，然后其他类继承这个类.

```

import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

class CommonMixin:
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    created_at: so.Mapped[Optional[datetime]] = so.mapped_column(
        default=datetime.now(timezone.utc),
        index=True
    )
    updated_at: so.Mapped[Optional[datetime]] = so.mapped_column(
        default=datetime.now(timezone.utc),
        onupdate=datetime.now(timezone.utc),
        index=True
    )

class User(CommonMixin, db.Model):
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)
    email: so.Mapped[str] = so.mapped_column(sa.String(120), index=True,
                                              unique=True)
    password: so.Mapped[Optional[str]] = so.mapped_column(sa.String(256))

```

此时 User 表会自动包含 id , created_at , updated_at 这三个字段，并将 id 作为主键。

S4.4. 关系模型

核心是 relationship 方法，所有参数都是可选。

- backref : 反向引用，可以省略单类的关系模式，但应优先考虑 back_populates .
- back_populates : 双向同步引用。
- secondary : 对于多对多关系用于指定中间表，并且是 Table 实例。
- primaryjoin : 连接父子表或是父表与关联表，通常通过外键和引用关系可以得到。
- secondaryjoin : 连接子表与关联表，通常通过外键和引用关系可以得到。
- cascade: str : 级联，确定如何关系父子级。默认为 "save-update, merge" , 更常用的方式是 "all, delete-orphan" . 更具体见 [sqlalchemy-cascade](#).
 - "all" : 表示 "save-update, merge, refresh-expire, expunge, delete" .
 - "save-update" : 对象保存到会话中时，关联项也会被添加到会话中。
 - "expunge" : 移除对象时将相关联的对象也从会话中移除 (session.expunge) .
 - "merge" : 会话合并 (session.merge) 时，会将关联对象也合并。

- "refresh-expire" : 不常用
- "delete" : 父项删除时, 子项也会删除, 否则是取消关联设定为 None .
- "delete-orphan" : 不止父项删除, 取消关联时也会删除子项, 这意味着子项只能有一个父对象, 可以在 python 端通过 `relationship(single_parent=True)` 验证.
- lazy :
 - True : 等同于 "select" .
 - False : 等同于 "joined" .
 - None : 等同于 "noload" .
 - "select" : 默认值. 查询父项时不会随带查询 (lazy-load), 只有在首次直接访问时使用查询所有项的方式获取, 可以通过 `TableName.rel` 获取.
 - "joined" : 查询父项时会使用 join 预加载 (eager-load) 所有相关项, 至于使用 [INNER] JOIN 还是 LEFT [OUTER] JOIN 取决于 relationship 的 innerjoin 参数 (默认是 False 即 LEFT [OUTER] JOIN).
 - "immediate" : 查询父项之后使用单独的 SELECT 语句查询预加载所有相关项.
 - "subquery" : 查询父项之后使用单独的 subquery 子查询预加载所有相关项.
 - "selectin" : 查询父项之后使用单独的 IN 子句查询预加载所有相关项.
 - "noload" : 任何时候都无法获取, 返回 None , **不推荐使用, 建议使用 `write_only_relationship` .**
 - "dynamic" : 无法直接访问, 返回 Select 对象, 可以进行进一步的过滤、排序等操作. 对应映射关系 DynamicMapped 会自动完成该配置.

现版本**已为遗留项**, 使用 `WriteOnlyMapped` / `write_only_relationship` **可以有更好的使用方法**.
 - "write_only" : 无法直接访问, 返回封装的 WriteOnlyCollection 对象, 具体见 WriteOnlyMapped
 - "raise" : 在未预加载的情况下无法获取, 报错形式返回.
 - "raise_on_sql" : 在未预加载的情况下无法通过 SQL 语句获取, 报错形式返回.
- innerjoin: bool = False : 连接方式, 默认外连接.
- single_parent: bool = False : 验证关系对应的父项唯一.
- active_history: bool : 默认 False, 访问时加载更改前的上一个值, 也适用于通过 `get_history()` 获取;

S4.4.1. 一对多 ORM

假设原有表 User 与 Post , 在**多**的一侧定义外键属性来关联两表.

```

import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

class User(CommonMixin, db.Model):
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)

class Post(CommonMixin, db.Model):
    body: so.Mapped[str] = so.mapped_column(sa.String(140))
    user_id: so.Mapped[int] = so.mapped_column(sa.ForeignKey(User.id), index=True)

```

这种方式在增加元组时需要实时刷新数据库，否则在创建 Post 元组时无法找到对应的 User. 通过 relationship 声明可以在增删改查时自动关联表.

```

# ...

class User(CommonMixin, db.Model):
    # ...
    posts: so.Mapped[List["Post"]] = so.relationship(back_populates='author')

class Post(CommonMixin, db.Model):
    # ...
    author: so.Mapped[User] = so.relationship(back_populates='posts')

```



```
author: so.Mapped[User] = so.relationship()
```

通过该定义直接就可以通过 `Post.author` 来访问对应的 `User` .

```

p=db.session.scalar(sa.select(Post).where(Post.author=u))
print(p.author)
# <User: 1; name: w>

```



```
author: so.Mapped[User] = so.relationship(backref='posts')
```

通过 back reference 可以自动在 `User` 类内增加一个属性 `posts` , 与 `Post.author` 相同能直接访问对应的 `Post` 列表.

```
u=db.session.scalar(sa.select(User).where(User.id=1))
print(u.posts)
# [<Post: 1; author: 1; body: text>, <Post: 2; author: 1; body: text>]
```

这种方法和给的范例代码效果一致.

- 这种方法无法更改 `posts` 关系模式的配置
- 没有显性得知 `User` 有 `posts` 属性.

所以更推荐范例代码.

 `posts: so.Mapped[List["Post"]]`

这里的 "Post" 是由于 `Post` 类是后定义的, 所以需要通过字符串参数的传递懒加载.

S4.4.2. 一对一 ORM

只需将一对多关系中的多项的关系使用非集合类型注释即可.

S4.4.3. 多对多 ORM

假设原有表 `User` 与 `Role`, 多对多关系需要创建关联表.

```
import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

user_role = sa.Table(
    "user_role",
    db.metadata,
    db.Column("user_id", sa.ForeignKey("user.id"), primary_key=True),
    db.Column("role_id", sa.ForeignKey("role.id"), primary_key=True)
)

class User(CommonMixin, db.Model):
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)

class Role(CommonMixin, db.Model):
    name: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                           unique=True)
```



sa.Table vs. class Table(db.Model)

由于这里是表示关联表，并非有属性的表，所以使用命令式定义，事实上使用 `class UserRole(db.Model)` 也可以，但这样是两个一对多关系，见 [关联对象](#)。

同样通过关系模式：


```

# ...

class User(CommonMixin, db.Model):
    # ...
    roles: so.Mapped[List["Role"]] = so.relationship(
        secondary=user_role,
        back_populates='users'
    )

class Role(CommonMixin, db.Model):
    # ...
    users: so.Mapped[List["User"]] = so.relationship(
        secondary=user_role,
        back_populates='roles'
    )

```

对于**自指多对多关系**，需要用 `primaryjoin` 与 `secondaryjoin` 明确指出关系模式查询的先后关联顺序。

```

import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

follow = sa.Table(
    "follow",
    db.metadata,
    db.Column("follower_id", sa.ForeignKey("user.id"), primary_key=True),
    db.Column("followed_id", sa.ForeignKey("user.id"), primary_key=True)
)

class User(CommonMixin, db.Model):
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)

    following: so.Mapped[List["User"]] = so.relationship(
        secondary=follow,
        primaryjoin=(follow.c.follower_id == id),
        secondaryjoin=(follow.c.followed_id == id),
        back_populates='followers'
    )

    followers: so.Mapped[List["User"]] = so.relationship(
        secondary=follow,
        primaryjoin=(follow.c.followed_id == id),
        secondaryjoin=(follow.c.follower_id == id),
        back_populates='following'
    )

```

S4.4.4. 关联对象

关联对象是多对多的一种变体，当关联表包含除父表和子表的外键之外的其他属性时，需要用到关联对象模式，只需将关联表 `Table` 实例使用映射类表示即可，转换成两个一对多关系。

```

import sqlalchemy as sa
import sqlalchemy.orm as so
from apps import db

class UserRole(db.Model):
    user_id: so.Mapped[int] = so.mapped_column(sa.ForeignKey("user.id"),
                                                primary_key=True)
    role_id: so.Mapped[int] = so.mapped_column(sa.ForeignKey("role.id"),
                                                primary_key=True)
    timestamp: so.Mapped[datetime] = so.mapped_column(
        default=lambda: datetime.now(timezone.utc),
        index=True
    )
    user_assoc: Mapped["User"] = so.relationship(back_populates='roles_assoc')
    role_assoc: Mapped["Role"] = so.relationship(back_populates='users_assoc')

class User(CommonMixin, db.Model):
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)

    roles_assoc: so.Mapped[List[UserRole]] = so.relationship(
        back_populates='user_assoc',
        viewonly=True
    )

    roles: so.Mapped[List["Role"]] = so.relationship(
        secondary='user_role',
        back_populates='users'
    )

class Role(CommonMixin, db.Model):
    name: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)

    users_assoc: so.Mapped[List[UserRole]] = so.relationship(
        back_populates='role_assoc',
        viewonly=True
    )

    users: so.Mapped[List[User]] = so.relationship(
        secondary='user_role',

```

```
back_populates='roles'  
)
```



viewonly

使用 `viewonly=True` 避免发生更改冲突.

S4.5. 映射类查询

对于 SQLAlchemy ORM 中的增删改查操作基本上都是先创建一个 python 内的 SQL 语句 `stmt`，或是自己编写对应的 SQL，再通过 `session.execute(stmt)` 执行.

`session.execute` 返回的类型会根据 `stmt` 的类型不同而不同. 对于 `Select` 对象，会返回 `<sqlalchemy.engine.result.ChunkedIteratorResult object>`，这是可迭代对象 (但不能用下标法访问).

```
stmt = sa.select(User) # Select 对象  
result = db.session.execute(stmt) # Result.ChunkedIteratorResult 对象  
for row in result:  
    print(row)
```

S4.5.0. Result 查询结果对象

- 对象关闭:

- `close()` -> `None`: 关闭 `result`. 正常情况下不主动使用.



部分方法会随带使用 `close` 方法

原因在于执行查询时会在会话中封装一个查询结果，这个结果一般都很大，如果多次调用**获取结果的方法**会占据很大内存，为了避免潜在的资源浪费，所以要关闭 `Result` 对象.

- `closed: bool`: 判断 `result` 是否关闭.

- 游标获取行:

- `all()` -> `List[Row[_T]]`: 等价于 `fetchall()`.
- `fetchall()` -> `List[Row[_T]]`: 返回结果中的所有行组成的列表. 调用后会减少所有行，游标指向空列表, 之后调用返回空列表.

- `fetchmany(size: int | None = None) -> List[Row[_T]]`: 返回指定数量的行组成的列表, 无参数默认 `all()`. 调用后会减少获取的行, 游标指向获取行之后的一行 (或是空列表).
- `fetchone() -> Row[_T] | None`: 返回一行. 调用后会减少获取的行, 游标指向下一行.

使用 `fetch_*`() 方法后调用 `first()` 会返回剩余行 (或为空)

```
result = db.session.execute(stmt)
# result: [
#   (id: 1, name: '123', email: '1.1'),
#   (id: 2, name: '222', email: '2.1')
# ]
print(result.fetchmany(1)) # [(1, '123', '1.1')]
print(result.first())      # (2, '222', '2.1')
```

• 获取单行:

- `first() -> Row[_T] | None`: 返回第一行或空. 调用后**关闭结果集**.
- `one() -> Row[_T]`: 返回一行, 如果找不到会引发 `NoResultFound` 错误, 如果还剩余多行则引发 `MultipleResultsFound` 错误. 调用后**关闭结果集**.
- `one_or_none() -> Row[_T] | None`: 返回一行, 如果找不到则返回 `None`, 如果还剩余多行则引发 `MultipleResultsFound` 错误. 调用后**关闭结果集**.

• 获取单行单值:

- `scalar() -> Any | none`: 返回第一行的第一列, 找不到则为 `None`. 调用后**关闭结果集**.
- `scalar_one() -> Any`: 返回一行的第一列, 如果找不到会引发 `NoResultFound` 错误, 如果还剩余多行则引发 `MultipleResultsFound` 错误. 调用后**关闭结果集**.
- `scalar_one_or_none() -> Any | none`: 返回一行的第一列, 如果找不到则返回 `None`, 如果还剩余多行则引发 `MultipleResultsFound` 错误. 调用后**关闭结果集**.

• 过滤获取行或值:

- `unique(strategy: Callable[[Any], Any] | None = None) -> Self`: 返回过滤之后的结果, 默认情况下就如同集合一般过滤相同的行.
- `columns(*col_expr: Literal[str, int, Column]) -> Self`: 返回结果中指定列组成的 `Result` 对象, 是对本身的引用. 其中 `col_expr` 可以是索引号、列名或是列 `Column` 对象.
- `scalars(index Literal[str, int, Column] = 0) -> ScalarResult[Any]`: 返回一个 `ScalarResult` 过滤对象, 该对象返回单个元素而非 `Row` 对象.



ScalarResult

也有方法

- all()
- fetchall()
- fetchmany(size)
- first()
- one()
- one_or_none()
- unique(strategy)
- partitions(size)

 要注意这种过滤返回结果是对本身的引用，是在同一地址上更改的

```
result = db.session.execute(stmt)
# result: [
#   (id: 1, name: '123', email: '1.1'),
#   (id: 2, name: '222', email: '2.1')
# ]
```

以下结果并非原本的 result：

```
ids = result.columns("id", 2)
# 列名为 `id` 的列和第三列，索引从 0 开始。
for row in result:
    print(row)
# (1, '1.1')
# (2, '2.1')
```

并且使用 ids 获取结果也会关闭 result：

```
ids = result.columns("id", 2)
print(ids.all())      # [(1, '1.1'), (2, '2.1')]
print(result.all())   # []
```

scalars 同理也会跟随：

```
print(result.scalars().all()) # [1, 2]
print(result.all())          # []
```

• 提取策略：

- `yield_per(num: int) -> Self`: 每次从数据库中提取 `num` 行数据, 在更改缓冲区剩余的数据, 这会影响 `fetchone()` 的结果. 在不声明该属性时数据库默认时一次性加载全部.

更改 `yield_per` 时的改变

```
result = db.session.execute(stmt)
# result: [
#   (id: 1, name: '123', email: '1.1'),
#   (id: 2, name: '222', email: '2.1'),
#   (id: 3, name: '333', email: '3.1'),
#   (id: 4, name: 'wyc', email: 'wyc.com'),
#   (id: 5, name: 'Jack', email: 'example.com')
# ]
```

对于加载前声明不会影响, 但加载后更改加载行数会清空缓冲区:

```
result.yield_per(3) # 声明一次加载 3 行
result.fetchone()  # 获取一行数据, 此时会在缓冲区加载 3 行数据
result.fetchone()
# (1, '123', '1.1')
# (2, '222', '2.1')
result.yield_per(2) # 改变加载行数, 会清空缓冲区剩余的行数 (剩余 1 行)
result.fetchone()  # 获取一行数据, 此时会在缓冲区加载 2 行数据
# (4, '333', '3.1')
```

对于默认情况以及 `yield_per <= 0` 时, 都是加载全部.

```
# result.yield_per(0)
# result.yield_per(-1)
result.fetchone()  # 获取一行数据, 此时会在缓冲区加载全部
# (1, '123', '1.1')
result.yield_per(2) # 清空缓冲区全部数据
result.fetchone()  # 获取一行数据, 此时数据库内已经没有数据了
# None
```

特殊情况下, 能够直接通过 `session.func` 查询而非使用 `session.execute(stmt).func` 获取结果, 其中包括:

- `session.get(entity, ident) -> Any | None` : 当只需要**通过主键**查询映射类表时可以使用, 找不到时返回 `None` .


```
db.session.get(User, 1)
db.session.get(User, {'id': 1})
db.session.get(MultiPriCls, (1, 2))
db.session.get(MultiPriCls, {'id1': 1, 'id2': 2})
# db.session.get(User, (1, 2))
# InvalidRequestError(只有一个 primary key 'User.id')
```

- `session.get_one(entity, indent) -> Any` : 与 `get()` 相同, 但在找不到时返回 `NoResultError` 错误.
- `session.scalar(statement)` : 如同 `session.execute(statement).scalar()`
- `session.scalars(statement)` : 如同 `session.execute(statement).scalars()`

S4.5.1. Select 选择对象

- **基本的构造** `Select` 对象的方法有:
 - `select` : 最基本的构造方法, 具体见下.
 - `table` .
 - `values(*columns, name: str | None = None, literal_binds: bool = False)` : 创建元组模板, 通常时使用 `values().data(Sequence[Any])` 的方式创建临时数据集.
- **集合复合查询** `CompoundSelect` 可以有以下方法, 这些方法也可以通过 `Select.func()` 的方式调用:
 - `union(*select)`
 - `union_all(*select)`
 - `intersect(*select)`
 - `intersect_all(*select)`
 - `except_(*select)`
 - `except_all(*select)`
- 同样也有**存在查询** `Exists` , 也可以通过 `Select.exists()` 调用:


```
exists(*columns)
```

 **使用 `exists` 方法和 `Select.exists` 创建 `Exists` 子句.**

```
exists_clause = sa.exists(table1.id).where(table1.col1 == table2.col2)
```



```
exists_clause = (
    sa.select(table1.id).where(table1.col1 == table2.col2).exists()
)
```

以上两种方法均可以

```
stmt = sa.select(table1.col1).where(exists_clause)
```

对于 `select` 方法，有几种构造方法：

- 选择单个实体: `select(User)`
- 选择多个实体: `select(User, Post)`，直接使用是笛卡尔积，通过 Joins [连接方法](#) 可以实现关联表。
- 选择属性: `select(User.username, User.email)`
- 附加属性: `select('Username: ' + User.username)`

Select 对象大多数都是通过链式方法来加入 SQL 语句：

- `select_stmt.from_statement(statement)`：可以在查询 `statement` 的基础上再进行过滤等操作。
- `select_stmt.where(*where_clause)`：内部可以使用简单的 SQL-python 语句，见 [where 子句](#)。
- `select_stmt.join_from().select_from()`：显式定义 FROM 子句和 JOIN 连接，见 [FROM 子句和 JOIN 连接](#)。
- `select_stmt.order_by(*attrs)`：排序查询，前后决定排序主次。升序/降序通过表属性的 `asc()` / `desc()` 方法实现。

```
print(select(User).order_by(User.username.desc()))
```

- `select_stmt.group_by(*attrs)`：分组子句，前后决定分组主次。
- `select_stmt.having(*having)`：分组后的筛选子句，可用语句同 `where`。
- `select_stmt.subquery()`：返回一个子查询，用于嵌套查询。见 [子查询和 CTE](#)。
- `select_stmt.cte()`：返回一个通用表表达式 (Common Table Expressions, CETs)，用于重复利用某一复用子查询，类似于子查询。见 [子查询和 CTE](#)。

S4.5.2. where 条件子句

例如 `User.id == 1` 实际上会产生 SQL 语句 `"user".id = :id_1` 而非简单的 `True / False`。允许以下方法：

- 条件运算符: `==`, `!=`, `>`, `<`, `>=`, `<=`

- 重复 where :

- 多次 where

```
select(User.id).where(User.username == 'wyc')
                    .where(User.email == 'wyc.com')

# SELECT user.id
# FROM user
# WHERE
#   user.username = :username_1
#   AND user.email = :email_1
```

- 单个 where 接收多个表达式

```
select(User.id).where(
    User.username == 'wyc',
    User.email == 'wyc.com'
)

# SELECT user.id
# FROM user
# WHERE
#   user.username = :username_1
#   AND user.email = :email_1
```

- 使用 and_() / or_() / not_()/~ 函数实现逻辑 AND / OR / NOT .

```
select(User.id).where(
    and_(
        ~or_(User.username == 'wyc', User.username == 'kurisu'),
        not_(User.email == 'example.com')
    )
)

# SELECT user.id
# FROM user
# WHERE
#   (user.name = :name_1 OR user.name = :name_2)
#   AND user.email = :email_1
```

- 列表属性的方法:

- in_(Iterable[Any]) : IN 查询.
 - is_(None) : IS NULL 查询
 - isnot(None) : IS NOT NULL 查询.
 - like(other, escape: str | None = None) : LIKE 查询. 其中 escape 表示转义字符.

- `ilike(other, escape: str | None = None)`: 同上 LIKE 查询, 不过不区分大小写.



SQLAlchemy 内的 LIKE 查询不包含 `[]` 与 `[^]` 通配符.



对特殊 LIKE 查询的封装

- `[i]startswith(other)`: 等价于 `[i]like(f'{other}%')`, 匹配以 `other` 开头.
- `[i]endswith(other)`: 等价于 `[i]like(f'%{other}')`, 匹配以 `other` 结尾.
- `[i]contains(other)`: 等价于 `[i]like(f'%{other}%')`, 匹配包含 `other`.

其中都有 `autoescape: bool = False` 作为参数, 表示自动将 `other` 内的 `&` / `_` 通配符及转义字符 `escape` 本身转义.

当未指定 `escape` 时开启 `autoescape=True` 会默认 `escape='/'`.

• 关系模式的方法:

- `contains(criterion)` 多侧关系使用可以查看是否包含某一单侧实例.

```
select(User.id).where(User.posts.contains(post))  
# 等价于 select(User.id).where(User.id == post.user_id)
```

- `any(criterion)` 多侧关系使用条件查看是否有至少一个成立

```
select(User.username).where(User.posts.any(Post.body.contains('1')))
```

- `has(criterion)` 单侧关系使用条件查看其属性是否满足条件

```
select(Post.id).where(Post.author.has(User.username == 'w'))
```



`filter_by(**kwargs)`

对于简单的“相等”关系, 可以使用如下形式:

```
select(User).filter_by(username='wyc', email='wyc.com')
```

S4.5.3. FROM 子句和 Joins 连接方式

上面讲到 `select(User, Post)` 的方式会使用笛卡尔积的形式查询，要创建关联表的 SQL 查询语句则需要使用 JOIN 连接 (使用 `where` 子句也能勉强达到相同目的，但可读性和规范性不能达到要求)

- `select_from(*from)` 可以指定 FROM 子句所引入的表.

两种 JOIN 方法:

- `join_from`: 显式定义左表与右表.

```
select_stmt.join_from(  
    from,  
    target,  
    on_clause: OnClause | None = None,  
    *,  
    isouter: bool = False  
    full: bool = False  
)
```

- `join`: 显示定义右表，左表自动推测.

```
select_stmt.join(  
    target,  
    on_clause: OnClause | None = None,  
    *,  
    isouter: bool = False  
    full: bool = False  
)
```

其中 `isouter=True` 代表使用 `LEFT OUTER JOIN`；`full=True` 代表使用 `FULL OUTER JOIN`。`full` 的优先级高于 `isouter`。

`on_clause` 是 ON 子句，正常情况下都会被自动推测.

使用 JOIN 连接表

```
print(select(User.username, Post.body).join(Post))
```

```
print(select(User.username, Post.body).join_from(User, Post))
```

```

print(select(User.username, Post.body).join_from(Post, User))

print(select(User.username, Post.body).join(Post, User.id == Post.user_id))

print(
    select(User.username, Post.body)
    .join_from(Post, User, User.id == Post.user_id)
)

```

以上几种形式均相同，都是以下 SQL 语句.

```

SELECT
    user.username, post.body
FROM
    user
    JOIN post ON user.id == post.user_id

```

另外，还可以直接通过关系模式连接，在关联时需要通过 `of_type()` **转换类型**（将关系模式转为对应映射表类），这在**两表间有多个关系模式时很重要**，需要明确指定是哪个关系关联.

```

print(select(User.username, Post.body).join(User.posts.of_type(Post)))

print(select(User.username, Post.body).join(Post.author.of_type(User)))

```

S4.5.4. 别名 alias 和标签 label

别名 alias

在同一张表被多次关联在同一关系内时需要使用别名来分别，`aliased` 项在 `sqlalchemy.orm` 模块下.

```

aliased(
    element,
    alias: FromClause | None = None,
    name: str | None = None,
    flat: bool = False,
    adapt_on_names: bool = False
)

```

直接使用

```
Author = so.aliased(User)
Follower = so.aliased(User)
print(
    sa.select(Post)
    .join(Post.author.of_type(Author))
    .join(Author.followers.of_type(Follower))
    .where(Follower.id == self.id)
    .order_by(Post.timestamp.desc())
)
```

从子句中别名

```
subq = (
    sa.select(User.username, User.email, Post.body)
    .join(Post)
    .subquery()
)
subq_alias = so.aliased(subq, name="subq_alias")
user_subq = so.aliased(User, subq_alias)
print(
    sa.select(user_subq).where(user_subq.email.endswith(".com"))
)
```

以上返回的 SQL 语句是

```
SELECT subq_alias.username, subq_alias.email
FROM
(
    SELECT
        "user".username AS username,
        post.body AS body
    FROM
        "user"
        JOIN post ON "user".id = post.user_id
) AS subq_alias
WHERE (subq_alias.email LIKE '%' || :email_1)
```

标签 label

直接查询 `select('Username: ' + User.username)` 会将 `'Username: ' + User.username` 命名成 `anon_<num>` .

可以通过

- `select(attr.label(label_name))`
- `select(label(label_name, attr))`

两种方式添加标签，如下例：

- `select(('Username: ' + User.username).label('un'))`
- `select(sa.label('un', 'Username: ' + User.username))`

都会创建以下 SQL 语句

```
SELECT "Username: " || user.username AS un
FROM user
```

通过使用标签可以使临时属性用在排序和排序上：

对于 `order_by` 和 `group_by` 方法，通过字符串文本传递参数均不会直接呈现，而是通过上下文找到匹配的表达式。

`asc()` 和 `desc()` 同样也可以。

```
stmt = (
    select(Address.user_id, func.count(Address.id).label("num_addresses"))
    .group_by("user_id")
    .order_by("user_id", desc("num_addresses"))
)
```

S4.5.5. 子查询和 CTE

- `subquery()` 子查询：

```

subq = (
    select(func.count(address.id).label("count"),
           address.user_id)
    .group_by(address.user_id)
    .subquery()
)
stmt = (
    select(user.name, user.fullname, subq.count)
    .join_from(user, subq)
)

```

- `cte()` 通用表表达式:

```

subq = (
    select(func.count(address.id).label("count"),
           address.user_id)
    .group_by(address.user_id)
    .cte()
)
stmt = (
    select(user.name, user.fullname, subq.count)
    .join_from(user, subq)
)

```

- `scalar_subquery()` 当子查询正好返回零行或一行或恰好一列时使用的子查询（大部分情况下是和聚合函数一起使用），返回的对象实数域 `ColumnElement` 对象 SQL 表达式层次结构，所以可以使用列表表达式例如

```

subq = (
    select(func.count(address.id))
    .where(user.id == address.user_id)
    .scalar_subquery()
)
print(subq == 5)

```

S4.5.6. SQL 函数

所有函数都在 `sqlalchemy.func` 内.

- 普通类:
 - `lower()`: 将字符串转换成小写
 - `concat()`: 连接多个字符串
 - `now()`: 返回当前时间

- `cast()` : 转换数据类型
- 聚合类:
 - `count()` : 计算返回的行数
 - `sum()` : 计算所有值的和
 - `max()` : 返回最大值
 - `min()` : 返回最小值

S4.6. 映射类行为

S4.6.1. 插入 INSERT

1. 创建新的类实例 `user = User(username='wyc', email='wyc@wyc.com')`
2. 通过 `db.session.add(user)` 将其添加到会话中



`session.new` 与 `session.flush()`

通过 `session.new` 可以查看当前会话中待处理的对象.

通过 `session.flush()` 可以将所有待处理对象访问数据库以刷新对象, 但不会提交事务 `session.commit()`, 当事务回滚 `session.rollback()` 时会撤销刷新的数据.

```
user = db.session.get(User, 1)
post = Post(body='another', author=user)
db.session.add(post)
db.session.new
# IdentitySet([<Post None: author(None), body(another)>])
db.session.flush()
db.session.new
# IdentitySet([])
db.session.rollback()
db.session.add(post)
db.session.new
# IdentitySet([<Post 4: author(1), body(another)>])
```

实际上并不需要手动刷新, 对于**关系模式**, 当通过 `user.posts` 查询 (或是直接查询 `Post`) 时也会查询到刚加入 `session` 的 `post`, 这是在读取时刷新了对象.

当遇到 `db.session.commit()` 或 `db.session.rollback()` 或 `db.session.close()` 时就会将会话中的任务按需求实施.

批量插入使用:

```
db.session.execute(
    sa.insert(User).values([
        {'username': 'user_1', 'email': 'user_1@example.com'},
        {'username': 'user_2', 'email': 'user_2@example.com'}
    ])
)
# 或是下面形式
db.session.execute(
    sa.insert(User),
    [
        {'username': 'user_1', 'email': 'user_1@example.com'},
        {'username': 'user_2', 'email': 'user_2@example.com'}
    ]
)
# 或是使用 db.session.add_all()
db.session.add_all([
    User(username='user_1', email='user_1@example.com'),
    User(username='user_2', email='user_2@example.com')
])
```



对更改对象使用 `returning` 以获取

通过 `session.scalars(insert().returning())` 可以在执行批量插入之后返回特定的值.

```
users = db.session.scalars(
    sa.insert(User).returning(User.id, User.username),
    [
        {'username': 'user_1', 'email': 'user_1@example.com'},
        {'username': 'user_2', 'email': 'user_2@example.com'}
    ]
)
for user in users:
    print(user)
```

当然这也适用于 UPDATE 和 DELETE.



配置 `render_nulls=True` 避免插入数据结构差别带来的分批插入

通过 `session.scalars(insert()).execution_options(render_nulls=True)` 可以在执行批量插入时避免分批插入，即插入数据结构差别带来的分批插入。

```
data = [  
    {'username': 'user_1', 'email': 'user_1@example.com', 'password': '12345678'},  
    {'username': 'user_2', 'email': 'user_2@example.com', 'password': '88888888'},  
    {'username': 'user_3', 'email': 'user_3@example.com'},  
    {'username': 'user_4', 'email': 'user_4@example.com', 'password': '12345678'}  
]
```

✗ 分批插入

```
db.session.execute(  
    sa.insert(User), data  
)
```

转换的 SQL 语句是

```
INSERT INTO user (username, email, password) VALUES (?, ?, ?)  
[...] [('user_1', 'user_1@example.com', '12345678'), ('user_2', 'user_2@example.com'  
INSERT INTO user (username, email) VALUES (?, ?)  
[...] ('user_3', 'user_3@example.com')  
INSERT INTO user (username, email, password) VALUES (?, ?, ?)  
[...] [('user_4', 'user_4@example.com', '12345678')]
```

✓ 避免分批插入

```
db.session.execute(  
    sa.insert(User).execution_options(render_nulls=True),  
    data  
)
```

转换的 SQL 语句是

```
INSERT INTO user (username, email, password) VALUES (?, ?, ?)
[...] [('user_1', 'user_1@example.com', '12345678'), ('user_2', 'user_2@example.com']
```

S4.6.2. 更新 UPDATE

1. 读取需要更新的数据组 `user = db.session.get(User, 1)`
2. 修改数据 `user.email = 'wyc@wyc.com'`, 此时是 `dirty` 数据
3. 将修改后的 `user` 添加到 `session` 中或是使用数据库查询刷新.

```
user = db.session.get(User, 1)
user.email = 'wyc@wyc.com'
user in db.session.dirty
# True
```

```
db.session.add(user)
# 或 db.session.flush()
# 或 db.session.scalars(sa.select(User))
user in db.session.dirty
# False
```

批量更新使用:

```
# 通过主键确定
db.session.execute(
    sa.update(User),
    [
        {'id': 1, 'username': 'user_1'},
        {'id': 2, 'email': 'user_2@example.com'}
    ]
)
# 通过 where 子句确定
db.session.execute(
    sa.update(User)
        .where(User.email.endswith('example.com'))
        .values(username='user_example')
)
```

S4.6.3. 删除 DELETE

1. 读取需要删除的数据 `user = db.session.get(User, 1)`
2. 通过 `db.session.delete(user)` 将其在 `session` 中标记为删除, 即 `not in session`

```
post = db.session.get(Post, 1)
post in db.session
# True
```

```
db.session.delete(post)
post in db.session
# True
```

```
或 db.session.flush()
# 或 db.session.scalars(sa.select(Post))
post in db.session
# False
```

批量删除使用:

```
db.session.execute(
    sa.delete(User).where(User.email.endswith('example.com'))
)
```

S4.7. 自定义 SQL 语句

S5. flask-WTF: 表单

S5.1. 创建表单

S6. flask-Login: 用户认证

Sx. 网页路由设计

Sy. 测试

映射模型的检测

```
mapper = User.__mapper__  
table = mapper.local_table  
columns = mapper.columns  
mapper.all_orm_descriptors.keys()
```

详见 `Mapper` , 使用 `inspect()` 也能检测 `Mapper` , 以及 `InstanceState`