

面试中的公共问题

面试中的公共问题

计算机基础

1. TCP/IP 模型相关问题。

建议阅读阮一峰的[《互联网协议入门（一）》](#)和[《互联网协议入门（二）》](#)。

2. HTTP 和 HTTPS 相关问题。

建议阅读阮一峰的[《HTTP 协议入门》](#)和[《SSL/TLS 协议运行机制的概述》](#)。

3. Linux 常用命令和服务。

4. 进程和线程之间的关系。什么时候用多线程？什么时候用多进程？。

5. 关系型数据库相关问题（ACID、事务隔离级别、锁、SQL 优化）。

6. 非关系型数据库相关问题（CAP/BASE、应用场景）。

Python 基础

7. 开发中用过哪些标准库和三方库。

标准库：sys / os / re / math / random / logging / json / pickle / shelve / socket / datetime / hashlib / configparser / urllib / itertools / collections / functools / threading / multiprocessing / timeit / atexit / abc / asyncio / base64 / concurrent.futures / copy / csv / operator / enum / heapq / http / profile / pstats / ssl / unittest / uuid

三方库：openpyxl / xlrd / xlwt / PyPDF2 / ReportLab / PyYAML / jieba / pillow / requests / urllib3 / responses / aiohttp / BeautifulSoup4 / lxml / pyquery / PyMySQL / psycopg2 / redis / PyMongo / Peewee / SQLAlchemy / alipay / PyJWT / itsdangerous / celery / flower / elasticsearch-dsl-py / PyCrypto / Paramiko / logbook / nose / pytest / coverage / Selenium / lineprofiler / memoryprofiler / matplotlib / pygal / OpenCV

8. 装饰器的作用、原理和实现。

9. 使用过哪些魔法方法。

建议阅读[《Python 魔术方法指南》](#)。

10. 生成式、生成器、迭代器的编写。

11. 列表、集合、字典的底层实现。

12. 垃圾回收相关问题。

13. 并发编程的相关问题。

14. 协程和异步 I/O 相关知识。

Django 和 Flask

15. MVC 架构（MTV）解决了什么问题。

16. 中间件的执行流程以及如何自定义中间件。

17. REST 数据接口如何设计（URL、域名、版本、过滤、状态码、安全性）。

建议阅读阮一峰的[《RESTful API 设计指南》](#)。

18. 使用 ORM 框架实现 CRUD 操作的相关问题。

- 如何实现多条件组合查询 / 如何执行原生的 SQL / 如何避免 N+1 查询问题

19. 如何执行异步任务和定时任务。

20. 如何实现页面缓存和查询缓存？缓存如何预热？

爬虫相关

21. Scrapy 框架的组件和数据处理流程。

22. 爬取的目的（项目中哪些地方需要用到爬虫的数据）。

23. 使用的工具（抓包、下载、清理、存储、分析、可视化）。

24. 数据的来源（能够轻松的列举出 10 个网站）。

25. 数据的构成（抓取的某个字段在项目中有何用）。

26. 反爬措施（限速、请求头、Cookie 池、代理池、Selenium、PhantomJS、RoboBrowser、TOR、OCR）。

27. 数据的体量（最后抓取了多少数据，多少 W 条数据或多少个 G 的数据）。

28. 后期数据处理（持久化、数据补全、归一化、格式化、转存、分类）。

数据分析

29. 科学运算函数库（SciPy 和 NumPy 常用运算）。

30. 数据分析库（Pandas 中封装的常用算法）。

31. 常用的模型及对应的场景（分类、回归、聚类）。

32. 提取了哪些具体的指标。

33. 如何评价模型的优劣。

34. 每种模型实际操作步骤，对结果如何评价。

项目相关

35. 项目团队构成以及自己在团队中扮演的角色（在项目中的职责）。

36. 项目的业务架构（哪些模块及子模块）和技术架构（移动端、PC 端、后端技术栈）。

37. 软件控制管理相关工具（版本控制、问题管理、持续集成）。

38. 核心业务实体及其属性，实体与实体之间的关系。

39. 用到哪些依赖库，依赖库主要解决哪方面的问题。

40. 项目如何部署上线以及项目的物理架构（Nginx、Gunicorn/uWSGI、Redis、MongoDB、MySQL、Supervisor 等）。

41. 如何对项目进行测试，有没有做过性能调优。

42. 项目中遇到的困难有哪些，如何解决的。

问题分析详述如下：

计算机基础：

01、TCP/IP 模型相关问题。

建议阅读阮一峰的《[互联网协议入门（一）](#)》和《[互联网协议入门（二）](#)》。

02、HTTP 和 HTTPS 相关问题。

建议阅读阮一峰的《[HTTP 协议入门](#)》和《[SSL/TLS 协议运行机制的概述](#)》。

03、Linux 常用命令和服务。

04、进程和线程之间的关系。什么时候用多线程？什么时候用多进程？。

05、关系型数据库相关问题（ACID、事务隔离级别、锁、SQL 优化）。

06、非关系型数据库相关问题（CAP/BASE、应用场景）。

计算机基础：

43. TCP/IP 模型相关问题。

建议阅读阮一峰的《[互联网协议入门（一）](#)》和《[互联网协议入门（二）](#)》。

44. HTTP 和 HTTPS 相关问题。

建议阅读阮一峰的《[HTTP 协议入门](#)》和《[SSL/TLS 协议运行机制的概述](#)》。

45. Linux 常用命令和服务。

46. 进程和线程之间的关系。什么时候用多线程？什么时候用多进程？。

解答：

进程：

优点：多进程可以同时利用多个 CPU，能够同时进行多个操作。

缺点：耗费资源（创建一个进程重新开辟内存空间）。

进程不是越多越好，一般进程个数等于 cpu 个数。

线程：

优点：共享内存，尤其是进行 IO 操作（网络、磁盘）的时候（IO 操作很少用 cpu），可以使用多线程执行并发操作。

缺点：抢占资源。

线程也不是越多越好，具体案例具体分析，切换线程关系到请求上下文切换耗时。

计算机中执行任务的最小单元：线程。

IO 密集型（不用 cpu）：多线程

计算密集型（用 cpu）：多进程

多线程和多进程是两个截然不同的概念。进程是内核分配给一个应用的相对独立的地址空间，有着自己的堆栈地址，当然是虚拟的，这样就保护了内核，使内核和应用隔离，个个具体的应用之间隔离，当然他们之间是可以通信的，这取决于操作系统。

线程是我们程序员最关注的，也就是我们在具体应用中的功能实现。分为主线程和子线程，在 Android 开发中我们称主线程为 UI 线程，在 JavaEE 和 Windows 开发中就是 main 函数等等，既然这样那么什么操作放在主线程中，什么操作放在子线程中，大致是这样，费时的操作和消耗资源也就是占内存的操作放在子线程中，更新 UI 之类的放在主线程中，他们之间通过消息传递实现。

像查询数据库、联网上传文件啊，断点续传和下载大数据文件啊等等操作都应该放在子线程中。

多线程的应用虽然提高了 CPU 的利用率，却也带来安全问题。熊掌和鱼不可兼得，如何处理效率和安全的问题，真的是考验一个程序员水

那么在 Python 中什么时候用多线程什么时候用多进程呢？当在 CPU-bound(计算密集型：绝大多数时间在计算)时最好用 - 多进程，而在 I/O bound(I/O 密集型：IO 处理 并且 大多时间是在等待)的时候最好用 - 多线程。

python 因为其全局解释器锁 GIL 而无法通过线程实现真正的平行计算。这个论断我们不展开，但是有个概念我们要说明，IO 密集型 vs. 计算密集型。

IO 密集型：读取文件，读取网络套接字频繁。

计算密集型：大量消耗 CPU 的数学与逻辑运算，也就是我们这里说的平行计算 c。

而 concurrent.futures 模块，可以利用 multiprocessing 实现真正的平行计算。

核心原理是：concurrent.futures 会以子进程的形式，平行的运行多个 python 解释器，从而令 python 程序可以利用多核 CPU 来提升执行速度。由于子进程与主解释器相分离，所以他们全局解释器锁也是相互独立的。每个子进程都能够完整的使用一个 CPU 内核。

平的现实问题。

47. 关系型数据库相关问题（ACID、事务隔离级别、锁、SQL 优化，隔离级别，脏读幻读，索引）。

ACID，是指数据库管理系统（DBMS）在写入或更新资料的过程中，为保证事务（transaction）是正确可靠的，所必须具备的四个特性：原子性（atomicity，或称不可分割性）、一致性（consistency）、隔离性（isolation，又称独立性）、持久性（durability）。

- 1.原子性 atomicity：在同一个事务内部的一组操作必须全部执行成功（或者全部失败）
 - 2.一致性 consistency：事务执行前后，数据完整性没有被破坏
 - 3.隔离性 isolation：一个事务操作的结果在何时以何种方式对其他并发的的事务操作可见
 - 4.持久性 durability：事务结束后，对数据的修改是永久的
- 2.如何保证 ACID
- 原子性由事务的 rundo 日志来保证。

持久性由事务的 redo 日志和来保证。

一致性由两阶段提交过程来保证。也即 redo log 和 binlog 的一致性的。

隔离性由 MVCC（MVCC 依赖的是 undo log 与 read view，仅在 RC、RR 中用到 MVCC）和锁来保证。

四种事务隔离级别：

读未提交：读取其他事务没有提交的数据(脏读，幻读,不可重复读，避免了第一类更新丢失)

读已提交：读已提交的数据(出现不可重复读，幻读)

可重复读：同一事务多次读取时能够保证所读取的数据一样，后续读取不能读到另一个事务已经提交的数据（会出现幻读）

串行读取：一个事务接一个事务串行执行

事务的隔离级别和存在的问题

- 每种隔离级别都规定了事务中所做的修改哪些是可见的，哪些是不可见的，较低级别的隔离可以执行更高的并发，系统开销也更低。
- 通过 **set xxx transaction isolation level read committed** 来设置隔离级别。

二、InnoDB 下的事务的隔离级别

查询 mysql 的事务隔离级别

SELECT @@tx_isolation

设置 mysql 的隔离级别

set session transaction isolation level REPEATABLE_READ

设置 mysql 的锁等待超时，单位是秒

set innodb_lockwait_timeout = 12000;

查看 mysql 的锁等待超时，单位是秒

show variables like 'innodb_lockwait_timeout';

事务隔离级别 名称 描述 解决了什么 未解决什么

READ_UNCOMMITTED 读未提交 是事务最低的隔离级别，它允许另外一个事务可以看到这个事务未提交的数据。

加锁方式：

- （1）事务读取数据不加锁。
- （2）事务在更新数据时，先对其加行级共享锁 S，直到事务结束才释放。

无脏读、不可重复读、幻读

READ_COMMITTED 读已提交

保证一个事务修改的数据提交后才能被另外一个事务读取。

该事务的隔离级别不光靠加锁实现，还要靠 MVCC，后面会讲。

- （1）事务在读取数据时，先对其加行级共享锁 S，一旦读完该行，立即释放该行级共享锁 S。
- （2）事务在更新数据时，先对其加行级排他锁 X，直到事务结束才释放。

脏读 不可重复读、幻读

REPEATABLE_READ 可重复读

保障可重复读的两种情况：

1.一致性非锁定读的情况：靠 MVCC 保障可重复读

select 执行的是快照读，读的是数据库记录的快照版本，是不加锁的。读的自己的快照，另外一个事务改了后，自己这边 **select** 出来的还是原来的，保证了可重复读。

2.锁定读的情况：靠锁保障可重复读

update、**delete**、**SELECT ... LOCK IN SHARE MODE**、**SELECT ... FOR UPDATE** 都是要加锁的

(1) 事务在读取数据时，先对其加行级共享锁 **S**，直到事务结束才释放。

(2) 事务在更新数据时，先对其加行级排他锁 **X**，直到事务结束才释放。

从该级别才开始加入间隙锁

脏读、不可重复读 幻读

SERIALIZABLE 串行

事务被处理为顺序执行

(1) 事务在读取数据时，先对其加表级共享锁，直到事务结束才释放。

(2) 事务在更新数据时，先对其加表级排他锁，直到事务结束才释放。

脏读、不可重复读、幻读 无

我在 **windows** 上安装的 **MySQL5.7**，它 **InnoDB** 默认事务隔离级别为可重复读（重复读取，**RR**）三、

脏读、不可重复读、幻读解释

1.脏读

读取到未提交事务修改的数据。

脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。

例子:

Mary 的原工资为 **1000**,财务人员将 **Mary** 的工资改为了 **8000**，但未提交事务

与此同时，**Mary** 正在读取自己的工资。**Mary** 发现自己的工资变为了 **8000**，欢天喜地！（脏读）

而财务发现操作有误，而回滚了事务,**Mary** 的工资又变为了 **1000**。

2.不可重复读

读取到已提交事务修改的数据。在一个事务中前后两次读取的结果并不致，导致了不可重复读。

例子:

在事务 1 中，**Mary** 读取了自己的工资为 **1000**,操作并没有完成。

在事务 2 中，这时财务人员修改了 **Mary** 的工资为 **2000**,并提交了事务。

在事务 1 中，**Mary** 再次读取自己的工资时，工资变为了 **2000**。

3.幻读

读取到新插入的数据。

第一个事务对一个表中的所有数据行进行了修改。同时，第二个事务向表中插入一行新数据。那么操作第一个事务的用户发现表中还有未修改的数据行。

例子:

目前工资为 **1000** 的员工有 **10** 人。

事务 1,读取所有工资为 1000 的员工。

这时事务 2 向 **employee** 表插入了一条员工记录,工资也为 1000

事务 1 再次读取所有工资为 1000 的员工 共读取到了 11 条记录四、

名词解释:

更新丢失: 每个事务不知道其它是无的操作, 后面的事务对数据做修改导致前面事务所做的数据修改丢失

解决办法: 对行加锁, 只允许一个更新事务。

脏读: 读取到其它事务没有提交的数据

幻读: 一个事务对数据库中所有数据做了修改, 这时新插入一条数据, 并提交, 这时在读取一次后面插入的数据被读出, 给人感觉有一条数据没有被修改

不可重复的: 一个事务多次读取的数据结果不一样

解决办法: 只有在事务已提交后才允许读取这条数据则可避免该问题

存在的问题:

脏读: 指某一事务读取到了另一事务未提交的脏数据, 违反了数据库的隔离性。解决方法: 采用读已提交的隔离级别。(脏数据和脏页是不同的概念, 脏页指的是缓冲池中被修改的页还没有刷新到磁盘中, 它是由于数据库和磁盘异步造成的, 最终刷新后依然能达到一致性, 而且异步操作还能提高性能。而脏数据指的是事务对缓冲池中行记录的修改还没有被提交, 导致一个事务读到另一个事务未提交的数据)

不可重复读: 指的是在一个事务内多次读取同一个数据, 而在这个期间, 另一个事务对这个数据执行了更新操作并提交了, 导致前一个事务两次读取的数据不一样, 违反了数据库的一致性。解决方法: 采用行级锁防止数据被修改。

幻读: 指事务在读取某个范围的数据时, 另一个事务又在该范围内插入了新的行, 导致后一次查询看到了前一次查询没看到的行。解决方法: 采用间隙锁, 对索引中的间隙进行锁定, 防止了幻影行的插入。

如何解决幻读?

MySQL 的 InnoDB 存储引擎默认的事务隔离级别是可重复读, 并且它有一个 MVCC 多版本并发控制的机制, 这个机制的原理是: 在每行记录的后面添加版本号, 在高并发场景下, 读操作读取的是数据之前的历史版本, 所以不会阻塞写操作, 而且就算别的事务又插入一个数据并立马提交, 新插入数据的版本号会比读取的版本号高, 所以读取的数据不变, 所以快照读不会出现幻读的问题了。当执行 **select** 操作时会默认执行快照读。对数据进行修改的操作采用的是当前读, 也就是会读到最新的数据, 为了避免幻读, 可以采用 **next-key locks** 来解决, **next-key locks** 由记录锁和间隙锁组成, 记录锁对索引进行加锁, 而间隙锁可以锁住数据行的间隙, 于是可以避免幻影行的插入, 解决了幻读。

MVCC (多版本并发控制)

MVCC 可以看做是行级锁的一个变种, 通过保存数据在某个时间点的快照来实现, 它尽量避免了解锁操作, 因此开销教低, 实现了非阻塞的读操作, 写操作也只锁定必要的行, 应对高并发事务, MVCC 比单纯的加锁更高效。

读未提交总是读取最新的数据行, 而不是符合当前事务版本的数据行, 而串行化则会对所有读取的行都加锁, 所以 MVCC 只在读已提交和可重复读这两个隔离级别下工作。InnoDB 的 MVCC 默认隔离级别是可重复读, 它是通过在每行记录后面保存两个隐藏的列来实现的。这两个列一个保存行的创建时间, 一个保存行的删除时间, 这里的时间指的是系统版本号, 每当修改数据时, 版本号加一。读操作通过 **Undo** 读取的是数据之前的历史版本, 所以不会阻塞写操作, 提高了数据库并发读写的性能。同时还可以解决幻读, 因

为读取事务时，读取的是 Undo 中旧版本的数据，这时就算另一个事务插入一个数据，并立马提交，新插入数据的版本号会比读取事务的版本号高，读取事务时所读的数据还是没变，解决了幻读。（更新时采用间隙锁，不仅锁定查询涉及到的行，还会对索引中的间隙进行锁定，防止了幻影行的插入）

事务的三级封锁协议

- 1.一级封锁协议：事务 T 中如果对数据 R 有写操作，必须在这个事务中对 R 的第一次读操作前对它加 X 锁，直到事务结束才释放事务结束包括正常结束（COMMIT）和非正常结束（ROLLBACK）。
- 2.二级封锁协议：一级封锁协议加上事务 T 在读取数据 R 之前必须先对其加 S 锁，读完后方可释放 S 锁。
- 3.三级封锁协议：一级封锁协议加上事务 T 在读取数据 R 之前必须先对其加 S 锁，直到事务结束才释放。

满足高级锁则一定满足低级锁。但有个非常致命的地方，一级锁协议就要在第一次读加 X 锁，直到事务结束。几乎就要在整个事务加写锁了，效率非常低。

三级封锁协议只是一个理论上的东西，实际数据库常用另一套方法来解决事务并发问题。

关系型数据库（五），数据库中的锁

目录

- 1.锁的分类
- 2.共享锁和排斥锁
- 3.乐观锁与悲观锁

五、数据库中的锁

1.锁的分类

- 按锁的粒度划分，可分为表级锁、行级锁、页级锁
- 按锁级别划分，可分为共享锁、排它锁
- 按加锁方式划分，可分为自动锁、显式锁
- 按操作划分，可分为DML锁、DDL锁
- 按使用方式划分，可分为乐观锁、悲观锁

2.共享锁和排斥锁

共享锁（读锁）

排斥锁（写锁）

共享锁和排斥锁的兼容性

	X	S
X	冲突	冲突
S	冲突	兼容

3.乐观锁与悲观锁

(1) 悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java 中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现。

(2) 乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 `write_condition` 机制，其实都是提供的乐观锁。在 Java 中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

[MySQL 优化十大技巧](<https://www.cnblogs.com/sharpest/p/10390035.html>)

MYSQL 优化主要分为以下四大方面：

设计：存储引擎，字段类型，范式与逆范式（为了建立冗余较小、结构合理的数据库，设计数据库时必须遵循一定的规则。在关系型数据库中这种规则就称为范式。范式是符合某一种设计要求的总结。要想设计一个结构合理的关系型数据库，必须满足一定的范式。第一范式 1NF，原子性

第二范式 2NF，消除部分依赖

第三范式 3NF，消除传递依赖）

功能：索引，缓存，分区分表。

架构：主从复制，读写分离，负载均衡。

合理 SQL：测试，经验。

Mysql 的sql 优化方法

MySQL 参数优化

1:MySQL 默认的最大连接数为 100，可以在 mysql 客户端使用以下命令查看

```
mysql> show variables like 'maxconnections';
```

2:查看当前访问 Mysql 的线程

```
mysql> show processlist;
```

3:设置最大连接数

```
mysql> set globle maxconnections = 5000;
```

最大可设置 16384,超过没用

4:查看当前被使用的 connections

```
mysql> show globle status like 'maxuserconnections'
```

对 MySQL 语句性能优化的经验

1、选择最合适的字段属性。2、尽量把字段设置为 NOT NULL。3、使用连接(JOIN)来代替子查询(Sub-Queries)。4、使用联合(UNION)来代替手动创建的临时表。5、事务。6、使用外键。7、锁定表。8、使用索引。9、优化的的查询语句：1 不使用子查询，2 避免函数索引，3 用 IN 来替换 OR，4 LIKE 双百分号无法使用到索引，5 读取适当的记录 LIMIT M,N，6 避免数据类型不一致，7 分组统计可以禁止排序，8 避免随机取记录，9 禁止不必要的 ORDER BY 排序，10 批量 INSERT 插入。

- ① 为查询缓存优化查询
- ② EXPLAIN 我们的 SELECT 查询(可以查看执行的行数)
- ③ 当只要一行数据时使用 LIMIT 1
- ④ 为搜索字段建立索引
- ⑤ 在 Join 表的时候使用相当类型的列，并将其索引
- ⑥ 千万不要 ORDER BY RAND ()
- ⑦ 避免 SELECT *
- ⑧ 永远为每张表设置一个 ID
- ⑨ 可以使用 ENUM 而不要 VARCHAR
- ⑩ 尽可能的使用 NOT NULL
- ⑪ 固定长度的表会更快
- ⑫ 垂直分割
- ⑬ 拆分打的 DELETE 或 INSERT 语句
- ⑭ 越小的列会越快
- ⑮ 选择正确的存储引擎
- ⑯ 小心 "永久链接"

MySQL 索引详细介绍

为什么要使用索引

索引是 MySQL 中一种十分重要的数据库对象。它是数据库性能调优技术的基础，常用于实现数据的快速检索。

索引就是根据表中的一列或若干列按照一定顺序建立的列值与记录行之间的对应关系表，实质上是一张描述索引列的列值与原表中记录行之间一一对应关系的有序表。

MySQL 数据库中的 B+树索引可以分为聚集索引和非聚集索引（辅助索引）

在 MySQL 中，通常有以下两种方式访问数据库表的行数据：

1) 顺序访问 2) 索引访问

索引的分类

注意：索引是在存储引擎中实现的，也就是说不同的存储引擎，会使用不同的索引。根据存储方式的不同，MySQL 中常用的索引在物理上分为以下两类。1) B-树索引，2) 哈希索引。

MyISAM 和 InnoDB 存储引擎：只支持 BTREE 索引，也就是说默认使用 BTREE，不能够更换
MEMORY/HEAP 存储引擎：支持 HASH 和 BTREE 索引

1、索引我们分为四类来讲 单列索引(普通索引，唯一索引，主键索引)、组合索引、全文索引、空间索引、

1.1、单列索引：一个索引只包含单个列，但一个表中可以有多个单列索引。这里不要搞混淆了。

1.1.1、普通索引：MySQL 中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。

1.1.2、唯一索引：索引列中的值必须是唯一的，但是允许为空值，

1.1.3、主键索引：是一种特殊的唯一索引，不允许有空值。

1.2、组合索引

在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时遵循最左前缀集合。这个如果还不明白，等后面举例讲解时在细说

1.3、全文索引

全文索引，只有在 MyISAM 引擎上才能使用，只能在 CHAR,VARCHAR,TEXT 类型字段上使用全文索引，介绍了要求，说说什么是全文索引，就是在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行，比如有"你是个靓仔，靓女 ..."通过靓仔，可能就可以找到该条记录。这里说的是可能，因为全文索引的使用涉及了很多细节，我们只需要知道这个大概意思，如果感兴趣进一步深入使用它，那么看下面测试该索引时，会给出一个博文，供大家参考。

1.4、空间索引

空间索引是对空间数据类型的字段建立的索引，MySQL 中的空间数据类型有四种，GEOMETRY、POINT、LINESTRING、POLYGON。在创建空间索引时，使用 SPATIAL 关键字。要求，引擎为 MyISAM，创建空间索引的列，必须将其声明为 NOT NULL。具体细节看下面

InnoDB 与 MyISAM 的区别

1. 存储结构

MyISAM 存储表分为三个文件 frm（表结构）、MYD（表数据）、MYI（表索引），而 InnoDB 如上文所说，根据存储方式不同，存储结构不同。

2. 事务支持

MyISAM 不支持事务，而 InnoDB 支持事务，具有事务、回滚和恢复的事务安全。

3. 外键和主键

MyISAM 不支持外键，而 InnoDB 支持外键。MyISAM 允许没有主键，但是 InnoDB 必须有主键，若未指定主键，会自动生成长度为 6 字节的主键。

4. 锁

MyISAM 只支持表级锁，而 Innodb 支持行级锁，具有比较好的并发性能，但是行级锁只有在 where 子句是对主键筛选才生效，非主键 where 会锁全表。

5. 索引

MyISAM 使用 B+树作为索引结构，叶节点保存的是存储数据的地址，主键索引 key 值唯一，辅助索引 key 可以重复，二者在结构上相同。Innodb 也是用 B+树作为索引结构，数据表本身就是按照 b+树组织，叶节点 key 值为数据记录的主键，data 域为完整的数据记录，辅助索引 data 域保存的是数据记录的主键。

MySQL 索引类型：

1、普通索引

最基本的索引，它没有任何限制，用于加速查询。

创建方法：

a. 建表的时候一起创建

```
CREATE TABLE mytable ( name VARCHAR(32) , INDEX indexmytablename (name) );
```

b. 建表后，直接创建索引

```
CREATE INDEX indexmytablename ON mytable(name);
```

c. 修改表结构

```
ALTER TABLE mytable ADD INDEX indexmytablename (name);
```

注：如果是字符串字段，还可以指定索引的长度，在列命令后面加上索引长度就可以了（例如:name(11)）

2、唯一索引

索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

创建方法：

a. 建表的时候一起创建

```
CREATE TABLE mytable ( name VARCHAR(32) , UNIQUE indexuniquemytable_name (name) );
```

b. 建表后，直接创建索引

```
CREATE UNIQUE INDEX indexmytablename ON mytable(name);
```

c. 修改表结构

```
ALTER TABLE mytable ADD UNIQUE INDEX indexmytablename (name);
```

注：如果是字符串字段，还可以指定索引的长度，在列命令后面加上索引长度就可以了（例如:name(11)）

3、主键索引

是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引。

创建方法：

a. 建表的时候一起创建

```
CREATE TABLE mytable ( id int(11) NOT NULL AUTO_INCREMENT , name VARCHAR(32) , PRIMARY KEY (id) );
```

b. 修改表结构

```
ALTER TABLE test.t1 ADD CONSTRAINT t1_pk PRIMARY KEY (id);
```

注：如果是字符串字段，还可以指定索引的长度，在列命令后面加上索引长度就可以了（例如:name(11)）

4、组合索引

指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合。

创建方法：

a. 建表的时候一起创建

```
CREATE TABLE mytable ( id int(11) , name VARCHAR(32) , INDEX indexmytableid_name (id,name) );
```

b. 建表后，直接创建索引

```
CREATE INDEX indexmytableid_name ON mytable(id,name);
```

c. 修改表结构

```
ALTER TABLE mytable ADD INDEX indexmytableid_name (id,name);
```

5、全文索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。

fulltext 索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的 where 语句的参数匹配。

fulltext 索引配合 match against 操作使用，而不是一般的 where 语句加 like。

它可以在 create table，alter table，create index 使用，不过目前只有 char、varchar，text 列上可以创建全文索引。

创建方法：

a. 建表的时候一起创建

```
CREATE TABLE article ( id int(11) NOT NULL AUTOINCREMENT, title char(250) NOT NULL, contents text NULL, `createat`int(10) NULL DEFAULT NULL , PRIMARY KEY (id`), FULLTEXT (contents) );
```

b. 建表后，直接创建索引

```
CREATE FULLTEXT INDEX indexarticlecontents ON article(contents);
```

c. 修改表结构

```
ALTER TABLE article ADD FULLTEXT INDEX indexarticlecontents (contents);
```

总结

虽然索引可以增加查询数据，但对于更新、创建或者删除的时候，需要去维护索引，导致性能会受影响，因此，索引也不能建立太多。

48. 非关系型数据库相关问题（CAP/BASE、应用场景）。

NoSQL 纪元

当下已经存在很多的 NoSQL 数据库，比如 MongoDB、Redis、Riak、HBase、Cassandra 等等。每一个都拥有以下几个特性中的一个：

- 不再使用 SQL 语言，比如 MongoDB、Cassandra 就有自己的查询语言
- 通常是开源项目
- 为集群运行而生
- 弱结构化——不会严格的限制数据结构类型

Nosql 数据库有哪些，分别适用什么场景

Redis：键值对数据库。适合缓存场景。

Mongodb：分布式文件存储数据库。适合字段变动很多的业务。如商品表的属性：冰箱的属性与 酒的属性 大部分不一致，用 JSON 存储在 Mongodb 中，有查询与运算

HBase：列式存储数据库。高压缩比，适用于统计类业务场景

Elasticsearch：全文检索数据库。弥补关系型数据库 多属性 like 效率很慢的情况。

•MongoDB

•特性：持久性更好\主从复制\查询利用 javascript 表达式\服务器端运行 javascript 函数\比 CouchDB 更容易就地升级\数据存储使用的是内存映射文件\数据库崩溃后需要对表进行修复\内置 Sharding

•最佳适用：如果你需要动态的查询，如果你更偏向与定义索引而非 map/reduce，如果你针对大数据库想要更好的性能，如果你想使用 CouchDB 而数据变化太快，磁盘不够用的话，可以使用 MongoDB。

•应用场景：一言以蔽之，MySQL 或 PostgreSQL 的替代品。

•Redis

•特性：内存数据库\2.0 版本之后可以部署到硬盘上\主从复制\简单的 Key-Value\集合、列表、散列结构\Value 可以设定过期

•最佳适用：在数据库大小可预见的前提下，适用于数据变化速度快的应用。

•应用场景：股价系统，数据分析，实时数据采集以及实时通信场景。

NoSQL 数据库的类型

NoSQL 可以大体上分为 4 个种类：**Key-value、Document-Oriented、Column-Family Databases 以及 Graph-Oriented Databases**。下面就一览这些类型的特性：

\一、键值（Key-Value）数据库**

键值数据库就像在传统语言中使用的哈希表。你可以通过 key 来添加、查询或者删除数据，鉴于使用主键访问，所以会获得不错的性能及扩展性。

\产品：**Riak、Redis、Memcached、Amazon's Dynamo、Project Voldemort****

有谁在使用：GitHub（Riak）、BestBuy（Riak）、Twitter（Redis 和 Memcached）、StackOverFlow（Redis）、Instagram（Redis）、Youtube（Memcached）、Wikipedia（Memcached）

适用的场景

储存用户信息，比如会话、配置文件、参数、购物车等等。这些信息一般都和 ID（键）挂钩，这种情况下键值数据库是个很好的选择。

不适用场景

1. 取代通过键查询，而是通过值来查询。Key-Value 数据库中根本没有通过值查询的途径。
2. 需要储存数据之间的关系。在 Key-Value 数据库中不能通过两个或以上的键来关联数据。
3. 事务的支持。在 Key-Value 数据库中故障产生时不可以进行回滚。

二、面向文档（Document-Oriented）数据库**

面向文档数据库会将数据以文档的形式储存。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名称与对应的值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用 XML、JSON 或者 JSONB 等多种形式存储。

产品：MongoDB、CouchDB、RavenDB

有谁在使用：SAP（MongoDB）、Codecademy（MongoDB）、Foursquare（MongoDB）、NBC News（RavenDB）

适用的场景

1. 日志。企业环境下，每个应用程序都有不同的日志信息。Document-Oriented 数据库并没有固定的模式，所以我们可以使用它储存不同的信息。
2. 分析。鉴于它的弱模式结构，不改变模式下就可以储存不同的度量方法及添加新的度量。

不适用场景

在不同的文档上添加事务。Document-Oriented 数据库并不支持文档间的事务，如果对这方面有需求则不应该选用这个解决方案。

三、列存储（Wide Column Store/Column-Family）数据库**

列存储数据库将数据储存在列族（column family）中，一个列族存储经常被一起查询的相关数据。举个例子，如果我们有一个 Person 类，我们通常会一起查询他们的姓名和年龄而不是薪资。这种情况下，姓名和年龄就会被放入一个列族中，而薪资则在另一个列族中。

产品：Cassandra、HBase

有谁在使用：Ebay（Cassandra）、Instagram（Cassandra）、NASA（Cassandra）、Twitter（Cassandra and HBase）、Facebook（HBase）、Yahoo!（HBase）

适用的场景

1. 日志。因为我们可以将数据储存在不同的列中，每个应用程序可以将信息写入自己的列族中。
2. 博客平台。我们储存每个信息到不同的列族中。举个例子，标签可以储存在一个，类别可以在一个，而文章则在另一个。

不适用场景

1. 如果我们需要 ACID 事务。Vassandra 就不支持事务。
2. 原型设计。如果我们分析 Cassandra 的数据结构，我们就会发现结构是基于我们期望的数据查询方式而定。在模型设计之初，我们根本不可能去预测它的查询方式，而一旦查询方式改变，我们就必须重新设计列族。

\四、图（Graph-Oriented）数据库**

图数据库允许我们将数据以图的方式储存。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple 和 Next，则会有两个“Founded by”的边将 Apple 和 Next 连接到 Steve Jobs。

产品：Neo4J、Infinite Graph、OrientDB

有谁在使用：Adobe（Neo4J）、Cisco（Neo4J）、T-Mobile（Neo4J）

适用的场景

1. 在一些关系性强的数据中
2. 推荐引擎。如果我们将数据以图的形式表现，那么将会非常有益于推荐的制定

不适用场景

不适合的数据模型。图数据库的适用范围很小，因为很少有操作涉及到整个图。

CAP（C: Consistency 一致性，A: Availability 可用性(指的是快速获取数据)，P: Tolerance of network Partition 分区容忍性(分布式)），BASE（Basically Available --基本可用，Soft-state --软状态/柔性事务（"Soft state" 可以理解为"无连接"的, 而 "Hard state" 是"面向连接"的），Eventual Consistency --最终一致性(最终一致性，也是是 ACID 的最终目的。)）和最终一致性（一言以蔽之：过程松，结果紧，最终结果必须保持一致性）是 NoSQL 数据库存在的三大基石。

BASE 模型反 ACID 模型，完全不同 ACID 模型，牺牲高一致性，获得可用性或可靠性：Basically Available 基本可用。支持分区失败(e.g. sharding 碎片划分数据库) Soft state 软状态 状态可以有一段不同步，异步。Eventually consistent 最终一致，最终数据是一致的就可以了，而不是时时一致。

BASE 思想的主要实现有

- 1.按功能划分数据库
- 2.sharding 碎片

BASE 思想主要强调基本的可用性，如果你需要高可用性，也就是纯粹的高性能，那么就要以一致性或容错性为牺牲，BASE 思想的方案在性能上还是有潜力可挖的。

非关系型数据库通常指数据以对象的形式存储在数据库中，而对象之间的关系通过每个对象自身的属性来决定

非关系数据库的（redis 和 MangDB）

为了处理海量数据，非关系数据库设计之初就是为了替代关系型数据库的关系

优点：

- 1.海量数据的增删改查是可以的
- 2.海量数据的维护和处理非常轻松

缺点：

- 1.数据和数据没有关系，他们之间就是单独存在的
- 2.非关系数据库没有关系，没有强大的事务关系，没有保证数据的完整性和安全性

适合处理海量数据，保证效率，不一定安全（统计数据，例如微博数据）

Python 基础

01、开发中用过哪些标准库和三方库。

标准库：sys / os / re / math / random / logging / json / pickle / shelve / socket / datetime / hashlib / configparser / urllib /
itertools / collections / functools / threading / multiprocessing / timeit / atexit / abc / asyncio / base64 / concurrent.futures / copy /
csv / operator / enum / heapq / http / profile / pstats / ssl / unittest / uuid

三方库：openpyxl / xlrd / xlwt / PyPDF2 / ReportLab / PyYAML / jieba / pillow / requests / urllib3 / responses / aiohttp /
BeautifulSoup4 / lxml / pyquery / PyMySQL / psycopg2 / redis / PyMongo / Peewee / SQLAlchemy / alipay / PyJWT / itsdangerous
/ celery / flower / elasticsearch-dsl-py / PyCrypto / Paramiko / logbook / nose / pytest / coverage / Selenium / lineprofiler /
memoryprofiler / matplotlib / pygal / OpenCV

02、装饰器的作用、原理和实现。

03、使用过哪些魔法方法：建议阅读《Python 魔术方法指南》。

04、生成式、生成器、迭代器的编写。

05、列表、集合、字典的底层实现。

06、垃圾回收相关问题。

07、并发编程的相关问题。

08、协程和异步 I/O 相关知识。

Python 基础：

1. 开发中用过哪些标准库和三方库。

标准库：sys / os / re / math / random / logging / json / pickle / shelve / socket / datetime / hashlib / configparser / urllib /
itertools / collections / functools / threading / multiprocessing / timeit / atexit / abc / asyncio / base64 / concurrent.futures / copy /
csv / operator / enum / heapq / http / profile / pstats / ssl / unittest / uuid

三方库：openpyxl / xlrd / xlwt / PyPDF2 / ReportLab / PyYAML / jieba / pillow / requests / urllib3 / responses / aiohttp /
BeautifulSoup4 / lxml / pyquery / PyMySQL / psycopg2 / redis / PyMongo / Peewee / SQLAlchemy / alipay / PyJWT / itsdangerous
/ celery / flower / elasticsearch-dsl-py / PyCrypto / Paramiko / logbook / nose / pytest / coverage / Selenium / lineprofiler /
memoryprofiler / matplotlib / pygal / OpenCV

2. 装饰器的作用、原理和实现。

python 装饰器之原理实现，作用，与例程

SpecYue 2018-12-03 18:06:47 506 收藏 1

展开

装饰器的功能

当需要对一段写好的代码添加一段新的需求的时候的时候我们就可以用装饰器实现。

```
def setfunc(func):
    def callfunc():
        print("———这是权限验证 1 ——")
        print("---这是权限验证 2 -----")
        func()
    return call_func
```

```
@setfunc
def test1():
    print("----test1----")
```

test1()

对 `test1` 函数添加验证 1 和验证 2 的功能，需要设计一个闭包，闭包的外部参数传递的是函数的引用，在内部函数里面添加需要添加的功能。比如以上这段代码，我们在 `test1` 函数前面添加上 `@setfunc` 这个装饰器，在调用 `test1` 函数的时候，我们就会按照 `callfunc` 函数里面的顺序执行。

```
zhangyue-OptiPlex-5050% python3 装饰器演示.py
———这是权限验证 1 ——
---这是权限验证 2 -----
----test1----
```

装饰器的原理

```
def set_func(func):
    def call_func():
        print("———这是权限验证 1 ——")
        print("---这是权限验证 2 -----")
        func()
    return call_func

#@set_func
def test_1():
    print("----test1----")

test_1 = set_func(test_1)

test_1()

#test_1()
```

装饰器的原理其实就是函数引用的传递，在闭包外部传递函数的引用，内部函数执行完“这是权限验证 1”和“这是权限验证 2”之后，就会把外部函数传递的函数引用参数拿过来执行

装饰器的原理其实就是函数引用的传递，在闭包外部传递函数的引用，内部函数执行完“这是权限验证 1”和“这是权限验证 2”之后，就会把外部函数传递的函数引用参数拿过来执行

```
zhangyue-OptiPlex-5050% python3 装饰器的实现过程.py
---这是权限验证 1 ---
---这是权限验证 2 ---
----test1----
```

如果把@setfunc这个装饰器去掉之后呢

```
zhangyue-OptiPlex-5050% python3 装饰器的实现过程.py
---这是权限验证 1 ---
---这是权限验证 2 ---
---这是权限验证 1 ---
---这是权限验证 2 ---
----test1----
```

不难理解，因为内部函数执行了两个验证之后，我们的函数的引用是被装饰器修饰过的，所以我们会在执行一下两个验证，最后再执行test1函数的功能

装饰器统计函数运行时间

```
import time
def setfunc(func):
    def callfunc():
        starttime=time.time()
        func()
        stoptime=time.time()
        print("运行时间是%f"%(starttime-stoptime))
        return callfunc
    @setfunc
    def test1():
        print("----test1----")
        for i in range(1000):
            pass
        test1()
```

对有参数无返回值的函数进行修饰

```
def setfunc(func):
    def callfunc(num):
        print("---这是权限验证 1 ---")
        print("---这是权限验证 2 ---")
        func(num)
        return callfunc
    @setfunc
    def test1(num):
```

```
print("----test1----%d"%num)
test1(10000)
```

对有参数无返回值的函数进行修饰的时候，修饰的函数有几个参数，闭包的内部函数就需要有几个参数，函数地址传递给外部函数，函数实参传递给内部参数

```
zhangyue-OptiPlex-5050% python3 对有参数无返回值的函数进行装饰.py
---这是权限验证1---
---这是权限验证2---
----test1----10000
```

不定长参数的函数装饰器

可以直接在闭包的内部函数里写不定长参数，当然也可以按照调用函数的形参格式进行传递。

```
def setfunc(func):
    print("开启装饰器")
    def callfunc(args,**kwargs):
        print("11111")
        print("22222")
        func(args,kwargs)
    return callfunc
@setfunc
def test(num,*args,kwargs):
    print("test---%d"%num)
    print("test---",args)
    print("test---",kwargs)
    test(100)
    test(100,200)
    test(100,200,300,mm=500)
```

以上就是按照不定长参数进行传递的。运行结果如下

```
zhangyue-OptiPlex-5050% python3 装饰器修饰不定长函数.py
开启装饰器
11111
22222
test---100
test--- ()
test--- {}
11111
22222
test---100
test--- (200,)
test--- {}
11111
22222
test---100
test--- (200, 300)
test--- {'mm': 500}
```

https://blog.csdn.net/qq_34788903

对带有返回值的函数进行装饰，通用装饰器

首先观察一段错误例程

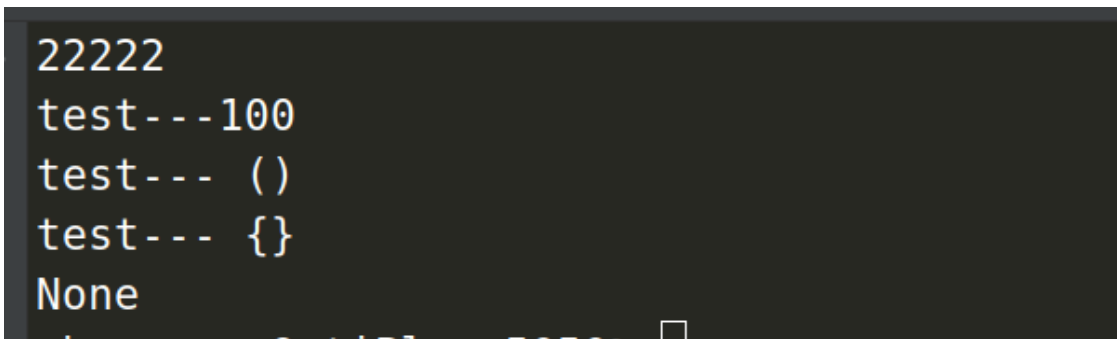
```
def setfunc(func):
    print("开启装饰器")
    def callfunc(args,**kwargs):
```

```

print("11111")
print("22222")
func(args,kwargs)
return callfunc
@setfunc
def test(num,*args,kwargs):
    print("test---%d"%num)
    print("test---",args)
    print("test---",kwargs)
    return "OK"
ret = test(100)
print(ret)

```

`test(100)`去调用闭包函数，执行完“开启装饰器 1 1 1 2 2 2”之后，去执行 `test` 函数，`test` 函数有返回值，但是 `call_func` 函数没有返回值，所以这个返回值并没有被 `ret` 接收到，所以打印出来的结果是 `None`



```

22222
test---100
test--- ()
test--- {}
None

```

正确的做法是 `call_func` 这个内部函数返回 `test` 函数的返回值，正确例程如下：

```

def setfunc(func):
    print("开启装饰器")
    def callfunc(args,**kwargs):
        print("11111")
        print("22222")
        return func(args,kwargs)
    return callfunc
@setfunc
def test(num,*args,kwargs):
    print("test---%d"%num)
    print("test---",args)
    print("test---",kwargs)
    return "OK"
ret = test(100)
print(ret)

```

那么如果被装饰函数没有返回值，但是装饰器有返回值会出现什么情况呢

```

def setfunc(func):
    print("开启装饰器")
    def callfunc(args,**kwargs):
        print("11111")
        print("22222")

```

```

return func(args,**kwargs)
return call_func

@set_func
def test(num,*args,**kwargs):
    print("test---%d"%num)
    print("test---",args)
    print("test---",kwargs)
    return "OK"

@set_func
def a():
    pass

ret = test(100)
print(ret)

ret2=a()
print(ret2)

```

这个时候 `ret2` 会返回 `None`，所有不会对函数有任何影响，故，以上例程的内部函数就是通用装饰器。

多个装饰器对同一个函数进行修饰

```

def setfunc(func):
    print("开启装饰器 1111")
    def callfunc():
        print("11111")
        return func()
    return call_func

def addqx(func):
    print("开启装饰器 222")
    def callfunc():
        print("222")
        return func()
    return call_func

@setfunc
@addqx
def test():
    print("----test----")

test()

```

首先程序是可以执行的，但是执行顺序呢，简单来说，开启装饰器的顺序是从下到上，执行内部函数的时候，由上到下，结果如下：

```

zhangyue-OptiPlex-5050% python3 多个装饰器对同一个函数进行装饰.py
开启装饰器 222
开启装饰器 1111
11111
222
----test----

```


类的装饰器

```
class Test(object):
def init(self,func):
self.func=func

def __call__(self, *args, **kwargs):
print("这里是装饰器添加的功能")
return self.func()
```

```
@Test
def get_str():
return "zhadf"
```

```
print(get_str())
```

运行结果：

```
zhangyue-OptiPlex-5050% python3 类的装饰器.py
这里是装饰器添加的功能
zhadf
zhangyue-OptiPlex-5050% █
```

装饰器带参数

```
def setlevel(levelnum):
def setfunc(func):
def callfunc(*args, **kwargs):
level = args[0]
```

```
if levelnum == 1:
print("权限验证 1 ")
elif levelnum == 2:
print("权限验证 2 ")
return func()
```

```
return call_func
```

```
return setfunc
```

```
@setlevel(1)
```

```
def test1():
print("test1")
return "OK"
```

```
@set_level(2)
```

```
def test2():
print("test2")
return "OK"
```

```
test1()
```

```
test2()
```

需要在闭包外层再定义一个函数，这个函数用来接受装饰器的参数。效果如上。

author:specyue@mail.ustc.edu.cn

github:https://github.com/zhangyuespec/mini_web/tree/master/mini_web 框架/装饰器

让你真正明白装饰器的工作原理和执行顺序

0. 什么是 Python 装饰器？

要弄明白什么是装饰器，装饰器是干什么？先看一个例子：装饰器的演变，所有的程序都是一步步迭代而来的，都是从冗余的结构不断优化，不断迭代而成最终的模块化代码的。从头往下看，让你彻底弄明白 python 装饰器的演变，执行顺序，多个装饰器执行顺序等工作原理。

#1. 定义一个函数，在不修改函数代码的前提下，对函数的功能进行拓展。比如权限验证。

```
def f1():  
    print("这里 f1 函数的功能展示")
```

#2. 定义一个高级函数（闭包）实现对 f1() 函数进行权限验证。

```
def fn(f1):  
    def fc():  
        print("这里开始对 f1 函数权限进行验证")  
        f1()  
        print("f1 函数已经处理完毕了")  
    return fc
```

#3. 实现：对函数调用，实现对 f1() 函数调用的权限验证。

'''结果如下：

这里开始对 f1 函数进行权限验证

这里 f1 函数的功能展示

f1 函数已经处理完毕了

'''

#4. 如果有多个修饰的函数的话，那上面函数调用麻烦了，需要一层层嵌套，比如：fn(f1)(). 结构臃肿。为了可视化和模块化，对上面的同一个功能的高级装饰函数进行统一标识，达到更好的效果。

```
def fn(f1):  
    def fc():  
        print("这里开始对 f1 函数权限进行验证")  
        f1()  
        print("f1 函数已经处理完毕了")  
    return fc  
@fn #这个@fn 标识符效果等同于 f1=fn(f1).  
def f1():  
    print("这里 f1 函数的功能展示")
```

注意这个时候函数不用再这样 fn(f1)() 的调用了，而是直接使用 f1() 即可达到 fn(f1)() 的效果了。

因为 @fn 的效果等同于 f1=fn(f1)，所以直接调用 f1() 相当于实现了 fn(f1)()，进而达到原来的效果。同样的效果但是代码就简化多了。

f1()

'''结果如下：

这里开始对 f1 函数进行权限验证

这里 f1 函数的功能展示

f1 函数已经处理完毕了

```
'''
```

通过上面的例子，我们可以得出：

1. 装饰器本质上是一个高级 Python 函数，通过给别的函数添加@标识的形式实现对函数的装饰

2. 装饰器的功能：它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

3. 面试题：什么是 python 装饰器？python 装饰器就是用于拓展原来函数功能的一种函数，这个函数的特殊之处在于它的返回值也是一个函数，使用 python 装饰器的好处就是在不用更改原函数的代码前提下给函数增加新的功能。

1. 单个装饰器的使用：装饰器的执行原理

#1. 定义两个装饰器（闭包函数，装饰函数都可以称呼），功能是给字体进行加粗和倾斜的标签。

```
def makeBold(fn):
    print("BBBBB" * 5)
    def wrapped():
        print("bbbbbb" * 5)
        return "" + fn() + ""
    return wrapped

def makeItalic(fn):
    print("IIIII" * 5)
    def wrapped():
        print("iiiiii" * 5)
        return "" + fn() + ""
    return wrapped
```

#2. 装饰器的使用，直接@加上函数名的形式，放到需要装饰的函数头上即可。

@makeBold #效果等同于 test_Bold=makeBold(test_Bold)，装饰器放在一个函数上，相当于将这个函数当成参数传递给装饰函数

```
def testBold():
    print("testBold" * 5)
    return "this is the test_Bold"
```

@makeItalic #效果等同于 test_Italic=makeItalic(test_Italic)，装饰器放在一个函数上，相当于将这个函数当成参数传递给装饰函数

```
def testItalic():
    print("testItali" * 5)
    return "this is the test_Italic"
```

下面实现对上面的单个装饰器代码的测试：

1.1 直接上面运行程序，什么不调用，也不操作，发现也有结果。

'''执行结果如下：

```
BBBBBBBBBBBBBBBBBBBBBBBBBBBBB
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII
'''
```

原因分析：

直接执行 python 程序,不调用任何方法,发现 makeBold 与 makeItalic 函数里面的第一个 print 函数也执行了。因为@makeBold 其效果等同于 test_Bold=makeBold(test_Bold),所以程序执行时,从上到下加载执行到函数体上方的标识符@makeBold@makeItalic 时,相当于执行了 test_Bold=makeBold(test_Bold),test_Italic=makeBold(test_Italic),所以相当于调用了 makeBold 和 makeItalic 这两个函数,所以依次执行了这两个函数中第一个 print 语句。因为@makeBold 在前,所以结果就是上面的 BBBB....和 IIIIIII....(其实这两个函数还有返回值,返回值分别是这两个函数内部的闭包,只是这里没有显示出来而已)

总结要点 1:

python 中装饰器是随着程序的加载运行而自动加载的,跟调不调用方法没有关系.所以只要是装饰器内部函数以外的部分都会自动加载执行,不用调用。

1.2.调用被装饰器装饰后的函数

#1.定义两个装饰器（闭包函数，装饰函数都可以称呼），功能是给字体进行加粗和倾斜的标签。

```
def makeBold(fn):
    print("BBBBB" * 5)
    def wrapped():
        print("bbbbbb" * 5)
        return "" + fn() + ""
    return wrapped

def makeItalic(fn):
    print("IIIII" * 5)
    def wrapped():
        print("iiiiii" * 5)
        return "" + fn() + ""
    return wrapped
```

#2.装饰器的使用,直接@加上函数名的形式,放到需要装饰的函数头上即可。

@makeBold #效果等同于 test_Bold=makeBold(test_Bold),装饰器放在一个函数上,相当于将这个函数当成参数传递给装饰函数

```
def testBold():
    print("testBold" * 5)
    return "this is the test_Bold"
```

@makeItalic #效果等同于 test_Italic=makeItalic(test_Italic),装饰器放在一个函数上,相当于将这个函数当成参数传递给装饰函数

```
def testItalic():
    print("testItali" * 5)
    return "this is the test_Italic"
```

#-----下面对函数进行调用-----

```
t = test_Bold() #调用 test_Bold()函数,相当于: makeBold(test_Bold)()
print(t) #打印 test_Bold 函数返回值
'''结果如下:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBB
IIIIIIIIIIIIIIIIIIIIIIIIIIII
bbbbbbbbbbbbbbbbbbbbbbbbbbbbb
test_Boldtest_Boldtest_Boldtest_Boldtest_Bold'''
```

```
<b>this is the test_Bold</b>
...
```

原因分析:

1.首先,程序执行后,从上到下加载,肯定是先加载到@makeBold@makeItalic 时,原理同上,这时候先把 BB BBBB....和 IIIIIIIII.....打印出来了。

2.因为@makeBold 其效果等同于 test_Bold=makeBold(test_Bold),所以这个时候程序在打印完 BBBB....以后,执行返回语句: return wrapped ;因为 wrapped 是个函数引用, 所以这个时候结果相当于 test_Bold=wrapped。即 test_Bold 指向闭包函数 wrapped。这个时候程序运行 t = test_Bold()执行时,等同于执行了 t=test_Bold()=wrapped()函数。所以这个时候执行了 wrapped 函数。先执行了 print("bbbbbb"*5)。打印了 bbbbbbbbbb(同理虽然@makeItalic 装饰的 test_Italic 函数的返回值也是对应的 wrapped 函数引用,但是因为后续没有调用 wrapped 函数,所以 wrapped 的函数内部没有执行。这里是难点,难点,难点)

3.接着往下执行 wrapped 函数的 return 语句,因为 return 语句里有调用了函数 test_Bold()。所以这个时候去执行 test_Bold()函数,所以执行了该函数内的“print("test_Bold"*5)”语句,打印了 test_Boldtest_Boldtest.....

4.这个时候 test_Bold()执行 return 语句,返回值是 this is the test_Bold, 在 wrapped 函数的 return " " + fn() + ""中作为参数使用。

5.所以 wrapped 的函数的返回值是: this is the test_Bold,最后将这个返回值,赋给 t,并且打印了 t。所以整个函数调用语句的结果就是如上。

总结要点 2:

1.装饰器是随着程序的执行而加载的,不是调用函数也会自动加载。

2.装饰器原理: @装饰器名 (@makeBold) 放在一个函数头上相当于将这个函数整体当做参数传递给这个装饰函数去执行,即等价于 test_Bold=makeBold(test_Bold),装饰器的使用大大简化了程序的代码。

2.多个装饰器同时修饰函数:装饰器执行顺序

#1.定义两个装饰函数,分别给字体进行加粗和倾斜的标签。

```
def makeBold(fn):
    print("BBBBB"5)
    def wrapped1(): #注意为了演示结果这里讲 wrapped 函数,分为 wrapped1,wrapped2
        print("bbbbbb"5)
        return "" + fn() + ""
    return wrapped1

def makeItalic(fn):
    print("IIIII"5)
    def wrapped2(): #注意为了演示结果这里讲 wrapped 函数,分为 wrapped1,wrapped2
        print("iiiiii" *3)
        return "" + fn() + ""
    return wrapped2
```

#2.使用两个装饰器同时装饰一个函数,可以三个,甚至多个。原理一样

@makeBold #注意 2.其效果等同于 testBI=makeBold(makeItalic(testBI))

@makeItalic #注意 1.其效果等同于 testBI=makeItalic(testBI)

```
def testBI():
    print("testBI"*5)
    return "this is the testBI"
```

下面实现对上面的两个装饰器代码的测试:

[illegible]

原因分析:

1.大家注意了，虽然@makeBold 写在了@makeItalic 的上面，但是结果显示，很明显先执行的是@makeItalic，即 makeItalic 函数时先加载执行的。所以当多个函数被多个装饰器装饰时，装饰器的加载顺序是从内到外的。其实很好理解：装饰器是给函数装饰的，所以要从靠近函数的装饰器开始从内往外加载。所以：打印的结果是 IIII III.....和 BBBB.....

`@makeBold` #注意2: 代码中的`@makeBold` 其效果 `test_B_I = makeBold(makeItalic(test_B_I))` , 即对下面 `makeItalic` 装饰后的结果进行装饰

```
@makeItalic #注意 1: :其效果等同于 test_B_I    =    makeItalic(test_B_I)
def test_B_I():
    print("test_B_I"*5)
    return "this is the test_B_I"
```

2.2.调用被两个装饰器装饰后的函数

#1. 定义两个装饰函数，分别给字体进行加粗和倾斜的标签。

```
def makeBold(fn):
    print("BBBBB" * 5)
    def wrapped1(): #注意为了演示结果这里讲 wrapped 函数，分为 wrapped1,wrapped2
        print("bbbbbb" * 5)
        return "" + fn() + ""
    return wrapped1

def makeItalic(fn):
    print("IIIII" * 5)
    def wrapped2(): #注意为了演示结果这里讲 wrapped 函数，分为 wrapped1,wrapped2
        print("iiiiii" * 3)
        return "" + fn() + ""
    return wrapped2
```

#2.使用两个装饰器同时装饰一个函数，可以三个，甚至多个。原理一样

```
@makeBold #注意 2.其效果等同于 testBI=makeBold( makeItalic(testBI) )
@makeItalic #注意 1.其效果等同于 testBI=makeItalic(testBI)
def testBI():
    print("testBI"*5)
    return "this is the testBI"
```

#-----注意下面对被两个装饰器修饰的函数进行调用-----

test B I() #调用被两个装饰器修饰后的函数 test B I()

```
print(test B I()) #打印 test B I 的返回值
```

'''结果如下:

```
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
iiiiiiiiiiiiiiiiiiii
test B Itest B Itest B It
```

```
<b><i>this is the test_B_I</i></b>
...
```

原因分析:

1.1.同理上面,因为装饰器是给函数装饰的,所以当一个函数被多个装饰器装饰时,装饰器的加载顺序是从内到外的,从下往上的,所以:打印的结果是 **IIIIIII.....**和 **BBBBB.....**

2.@makeBold #注意 2: 其效果 test_B_I = makeBold(makeItalic(test_B_I)), 即对下面 makeItalic 装饰后的结果进行装饰

@makeItalic #注意 1: :其效果等同于 test_B_I = makeItalic(test_B_I)

```
def test_B_I():
    print("test_B_I"*5)
    return "this is the test_B_I"
```

3.注意上面代码,多个装饰器时,装饰器是从内往外加载。所以先是@makeItalic 装饰 test_B_I 函数,其效果等同于 test_B_I = makeItalic(test_B_I)。返回值是其内部的 wrapped2 函数引用。@makeItalic 加载执行后的结果是: test_B_I=wrapped2。这个时候@makeBold 装饰器再对这个结果(函数)进行装饰。所以这时候@makeBold 装饰效果等价于:

test_B_I = makeBold(makeItalic(test_B_I)), 也等价于 test_B_I=makeBold(wrapped2)。又因为 makeBold()的返回值是 wrapped1。即 makeBold(wrapped2)= wrapped1。所以最后@makeBold@makeItalic 装饰后的结果时 test_B_I=wrapped1。只是在 wrapped1 里面调用了 wrapped2。所以,当执行函数调用语句 test_B_I()时,相当于执行了 wrapped1()。这个时候打印了 wrapped1 函数内部的 print("bbbbbb"*5),所以接着结果时: bbbbbbbbbbbbbbb bbbbbbbbbbbbbbb

4.紧接着,执行 wrapped1 函数的 return 语句: return "" + fn() + "".因为@makeBold 装饰的效果等价于 makeBold(makeItalic(test_B_I)), 即 makeBold 装饰的函数是@makeItalic 装饰后的结果。等价于 makeBold(wrapped2)。所以实际执行的是: return "" +wrapped2() + "".故这个时候要调用 wrapped2 函数。执行了 print("iiiiiii" *3)。所以接着打印了结果是: iiiiiiiiiiiiiiiiiii

5.接着执行 wrapped2 里面的 return "<i>" + fn() + "</i>".因为 wrapped2 装饰的是 test_B_I 函数,所以这里 fn()=test_B_I()。这个时候又去调用 test_B_I()函数,所以执行了 print("test_B_I"*5)。打印了: test_B_Itest_B_Itest_B_Itest_B_Itest_B_I

6.因为 test_B_I 的返回值是 this is the test_B_I。所以 wrapped2 的返回值是<i>this is the test_B_I</i>。所以 wrapped1 的返回值是: <i>this is the test_B_I</i>。所以最后被两个装饰器装饰后的 test_B_I()函数的返回值结果是: <i>this is the test_B_I</i>。所以最后整个函数调用的结果如上图。

关于多个装饰器修饰一个函数总结要点:

1.当一个函数被多个装饰器装饰时,装饰器的加载顺序是从内到外的(从下往上的)。其实很好理解:装饰器是给函数装饰的,所以要从靠近函数的装饰器开始从内往外加载

2.外层的装饰器,是给里层装饰器装饰后的结果进行装饰。相当于外层的装饰器装饰的函数是里层装饰器的装饰原函数后的结果函数(装饰后的返回值函数)。

统一声明:关于原创博客内容,可能会有部分内容参考自互联网,如有原创链接会声明引用;如找不到原创链接,在此声明如有侵权请联系删除哈。关于转载博客,如有原创链接会声明;如找不到原创链接,在此声明如有侵权请联系删除哈。

3. 使用过哪些魔法方法: (建议阅读《Python 魔法方法指南》)。

> 建议阅读[《Python 魔法方法指南》](<https://pycoders-weekly-chinese.readthedocs.io/en/latest/issue6/a-guide-to-pythons-magic-methods.html>)。

>

> # python 的常用魔法方法详细总结

>

> # **阅读目录**

>

> 1. [构造和初始化](https://www.cnblogs.com/zhouyixian/p/11129347.html#_label0)

> 2. [属性访问控制](https://www.cnblogs.com/zhouyixian/p/11129347.html#_label1)

> 3. [描述器对象](https://www.cnblogs.com/zhouyixian/p/11129347.html#_label2)

3) > 4. [构造自定义容器(Container)](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label1)

> 5. [上下文管理](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label4)

> 6. [对象的序列化](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label5)

> 7. 运算符相关的魔术方法

> - [比较运算符](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label6_0)

> - [一元运算符和函数](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label6_1)

> - [算术运算符](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label6_2)

> - [反算术运算符](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label6_3)

> - [增量赋值](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label6_4)

> - [类型转化](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label6_5)

> 8. [其他魔术方法](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label7)

> 9. [Python3 中的差异](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_label8)

>

>

>

> [回到顶部](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_labelTop)

>

> ## 构造和初始化

>

> `__init__` 我们很熟悉了,它在对象初始化的时候调用,我们一般将它理解为"构造函数".

>

> 实际上,当我们调用`x = SomeClass()`的时候调用,`__init__`并不是第一个执行的,`__new__`才是。所以准确来说,是`__new__`和`__init__`共同构成了"构造函数".

>

> `__new__`是用来创建类并返回这个类的实例,而`__init__`只是将传入的参数来初始化该实例。

>

> `__new__`在创建一个实例的过程中必定会被调用,但`__init__`就不一定,比如通过`pickle.load`的方式反序列化一个实例时就不会调用`__init__`。

>

> `__new__`方法总是需要返回该类的一个实例,而`__init__`不能返回除了`None`的任何值。比如下面例子:

>

> ...

```
> class Foo(object):
>     def __init__(self):
>         print 'foo __init__'
>         return None # 必须返回 None, 否则抛 TypeError
>
>     def __del__(self):
>         print 'foo __del__'
> ...
```

>

> 实际中,你很少会用到`__new__`,除非你希望能够控制类的创建。

> 如果要讲解`__new__`,往往需要牵扯到`metaclass`(元类)的介绍。

>

> 对于`__new__`的重载,[Python 文档中](https://www.python.org/download/releases/2.2/descrintro/#__new__)也有了详细的介绍。

>

> 在对象的生命周期结束时,`__del__`会被调用,可以将`__del__`理解为"析构函数".

> `__del__`定义的是当一个对象进行垃圾回收时候的行为。

>

> 有一点容易被人误解,实际上,`x.__del__()`并不是对于`del x`的实现,但是往往执行`del x`时会调用

```
`x.__del__()`.
```

```
>
```

```
> 怎么来理解这句话呢？继续用上面的 Foo 类的代码为例：
```

```
>
```

```
> ```
```

```
> foo = Foo()
```

```
> foo.__del__()
```

```
> print foo
```

```
> del foo
```

```
> print foo # NameError, foo is not defined
```

```
> ```
```

```
>
```

```
> 如果调用了`foo.__del__()`，对象本身仍然存在。但是调用了`del foo`，就再也没有 foo 这个对象了。
```

```
>
```

> 请注意，如果解释器退出的时候对象还存在，就不能保证`__del__`被确切的执行了。所以`__del__`并不能替代良好的编程习惯。

```
> 比如，在处理 socket 时，及时关闭结束的连接。
```

```
>
```

```
> [回到顶部](https://www.cnblogs.com/zhoudyixian/p/11129347.html#_labelTop)
```

```
>
```

```
> ## 属性访问控制
```

```
>
```

> 总有人要吐槽 Python 缺少对于类的封装，比如希望 Python 能够定义私有属性，然后提供公共可访问的 `getter` 和 `setter`。Python 其实可以通过魔术方法来实现封装。

```
>
```

```
> ```
```

```
> __getattr__(self, name)
```

```
> ```
```

```
>
```

> 该方法定义了你试图访问一个不存在的属性时的行为。因此，重载该方法可以实现捕获错误拼写然后进行重定向，或者对一些废弃的属性进行警告。

```
>
```

```
> ```
```

```
> __setattr__(self, name, value)
```

```
> ```
```

```
>
```

```
> `__setattr__` 是实现封装的解决方案，它定义了你为属性进行赋值和修改操作时的行为。
```

> 不管对象的某个属性是否存在，它都允许你为该属性进行赋值，因此你可以为属性的值进行自定义操作。有一点需要注意，实现`__setattr__`时要避免“无限递归”的错误，下面的代码示例中会提到。

```
>
```

```
> ```
```

```
> __delattr__(self, name)
```

```
> ```
```

```
>
```

> `__delattr__`与`__setattr__`很像，只是它定义的是你删除属性时的行为。实现`__delattr__`是同时要避免“无限递归”的错误。

```
>
```

```
> ```
```

```
> __getattribute__(self, name)
```

```
> ```
```

```
>
```

> `__getattribute__`定义了你的属性被访问时的行为，相比较，`__getattr__`只有该属性不存在时才会起作用。

```
> 因此，在支持`__getattribute__`的 Python 版本，调用`__getattr__`前必定会调用`__getattribute__`
```

。`__getattr__`同样要避免"无限递归"的错误。

> 需要提醒的是，最好不要尝试去实现`__getattr__`，因为很少见到这种做法，而且很容易出 bug。

>

> 例子说明`__setattr__`的无限递归错误：

>

> ```

> def __setattr__(self, name, value):

> self.name = value

> # 每一次属性赋值时，__setattr__都会被调用，因此不断调用自身导致无限递归了。

> ```

>

> 因此正确的写法应该是：

>

> ```

> def __setattr__(self, name, value):

> self.__dict__[name] = value

> ```

>

> `__delattr__`如果在其实现中出现`del self.name`这样的代码也会出现"无限递归"错误，这是一样的原因。

>

> 下面的例子很好的说明了上面介绍的 4 个魔术方法的调用情况：

>

> ```

> class Access(object):

>

> def __getattr__(self, name):

> print '__getattr__'

> return super(Access, self).__getattr__(name)

>

> def __setattr__(self, name, value):

> print '__setattr__'

> return super(Access, self).__setattr__(name, value)

>

> def __delattr__(self, name):

> print '__delattr__'

> return super(Access, self).__delattr__(name)

>

> def __getattribute__(self, name):

> print '__getattribute__'

> return super(Access, self).__getattribute__(name)

>

> access = Access()

> access.attr1 = True # __setattr__调用

> access.attr1 # 属性存在,只有__getattribute__调用

> try:

> access.attr2 # 属性不存在,先调用__getattribute__,后调用__getattr__

> except AttributeError:

> pass

> del access.attr1 # __delattr__调用

> ```

>

> [回到顶部](https://www.cnblogs.com/zhouyixian/p/11129347.html#_labelTop)

>

> ## 描述器对象

>
> 我们从一个例子来入手,介绍什么是描述符,并介绍`__get__`,`__set__`,`__delete__`的使用。(放在这里介绍是为了跟上一小节介绍的魔术方法作对比)

>
> 我们知道,距离既可以用单位"米"表示,也可以用单位"英尺"表示。
> 现在我们定义一个类来表示距离,它有两个属性:米和英尺。

```
>
>
> class Meter(object):
>     '''Descriptor for a meter.'''
>     def __init__(self, value=0.0):
>         self.value = float(value)
>     def __get__(self, instance, owner):
>         return self.value
>     def __set__(self, instance, value):
>         self.value = float(value)
>
> class Foot(object):
>     '''Descriptor for a foot.'''
>     def __get__(self, instance, owner):
>         return instance.meter * 3.2808
>     def __set__(self, instance, value):
>         instance.meter = float(value) / 3.2808
>
> class Distance(object):
>     meter = Meter()
>     foot = Foot()
>
> d = Distance()
> print d.meter, d.foot # 0.0, 0.0
> d.meter = 1
> print d.meter, d.foot # 1.0 3.2808
> d.meter = 2
> print d.meter, d.foot # 2.0 6.5616
>
>
>
```

> 在上面例子中,在还没有对 `Distance` 的实例赋值前,我们认为 `meter` 和 `foot` 应该是各自类的实例对象,但是输出却是数值。这是因为`__get__`发挥了作用。

>
> 我们只是修改了 `meter`,并且将其赋值成为 `int`,但 `foot` 也修改了。这是`__set__`发挥了作用。

> 描述器对象(`Meter`、`Foot`)不能独立存在,它需要被另一个所有者类(`Distance`)所持有。

> 描述器对象可以访问到其拥有者实例的属性,比如例子中 `Foot` 的`instance.meter`。

>
> 在面向对象编程时,如果一个类的属性有相互依赖的关系时,使用描述器来编写代码可以很巧妙的组织逻辑。

> 在 `Django` 的 ORM 中, `models.Model` 中的 `IntegerField` 等,就是通过描述器来实现功能的。

>
> 一个类要成为描述器,必须实现`__get__`,`__set__`,`__delete__`中的至少一个方法。下面简单介绍下:

```
>
>
>
> __get__(self, instance, owner)
```

>
>
> 参数 `instance` 是拥有者类的实例。参数 `owner` 是拥有者类本身。`__get__`在其拥有者对其读值的时候调

用。

```
>
> ...
> __set__(self, instance, value)
> ...
>
> `__set__` 在其拥有者对其进行修改值的时候调用。
>
> ...
> __delete__(self, instance)
> ...
>
> `__delete__` 在其拥有者对其进行删除的时候调用。
>
> [回到顶部](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_labelTop)
>
> ## 构造自定义容器(Container)
>
> 在 Python 中, 常见的容器类型有: dict, tuple, list, string。
> 其中 tuple, string 是不可变容器, dict, list 是可变容器。
> 可变容器和不可变容器的区别在于, 不可变容器一旦赋值后, 不可对其中的某个元素进行修改。
> 比如定义了 `l = [1, 2, 3]` 和 `t = (1, 2, 3)` 后, 执行 `l[0] = 0` 是可以的, 但执行 `t[0] = 0` 则会
报错。
>
> 如果我们要自定义一些数据结构, 使之能够跟以上的容器类型表现一样, 那就需要去实现某些协议。
>
> 这里的协议跟其他语言中所谓的"接口"概念很像, 一样的需要你去实现才行, 只不过没那么正式而已。
>
> 如果要自定义不可变容器类型, 只需要定义 `__len__` 和 `__getitem__` 方法;
> 如果要自定义可变容器类型, 还需要在不可变容器类型的基础上增加定义 `__setitem__` 和 `__delitem__`
、。
>
> 如果你希望你的自定义数据结构还支持"可迭代", 那就还需要定义 `__iter__`。
>
> ...
> __len__(self)
> ...
>
> 需要返回数值类型, 以表示容器的长度。该方法在可变容器和不可变容器中必须实现。
>
> ...
> __getitem__(self, key)
> ...
>
> 当你执行 `self[key]` 的时候, 调用的就是该方法。该方法在可变容器和不可变容器中也都必须实现。
> 调用的时候, 如果 key 的类型错误, 该方法应该抛出 TypeError;
> 如果没法返回 key 对应的数值时, 该方法应该抛出 ValueError。
>
> ...
> __setitem__(self, key, value)
> ...
>
> 当你执行 `self[key] = value` 时, 调用的是该方法。
>
> ...
```

```

> __delitem__(self, key)
> ...
>
> 当你执行`del self[key]`的时候,调用的是该方法。
>
> ...
> __iter__(self)
> ...
>
> 该方法需要返回一个迭代器(iterator)。当你执行`for x in container:` 或者使用`iter(container)`
`时,该方法被调用。
>
> ...
> __reversed__(self)
> ...
>
> 如果想要该数据结构被内建函数`reversed()`支持,就还需要实现该方法。
>
> ...
> __contains__(self, item)
> ...
>
> 如果定义了该方法,那么在执行`item in container` 或者 `item not in container`时该方法就会被
调用。
> 如果没有定义,那么 Python 会迭代容器中的元素来一个一个比较,从而决定返回 True 或者 False。
>
> ...
> __missing__(self, key)
> ...
>
> `dict`字典类型会有该方法,它定义了`key`如果在容器中找到时触发的行为。
> 比如`d = {'a': 1}`,当你执行`d[notexist]`时,`d.__missing__('notexist')`就会被调用。
>
> 下面举例,使用上面讲的魔术方法来实现 Haskell 语言中的一个数据结构。
>
> ...
> # -*- coding: utf-8 -*-
> classFunctionalList:
>     ''' 实现了内置类型 list 的功能,并丰富了一些其他方法: head, tail, init, last, drop, take'
..
>
>     def__init__(self, values=None):
>         if values is None:
>             self.values = []
>         else:
>             self.values = values
>
>     def__len__(self):
>         return len(self.values)
>
>     def__getitem__(self, key):
>         return self.values[key]
>
>     def__setitem__(self, key, value):
>         self.values[key] = value

```

```

>
> def __delitem__(self, key):
>     del self.values[key]
>
> def __iter__(self):
>     return iter(self.values)
>
> def __reversed__(self):
>     return FunctionalList(reversed(self.values))
>
> def append(self, value):
>     self.values.append(value)
> def head(self):
>     # 获取第一个元素
>     return self.values[0]
> def tail(self):
>     # 获取第一个元素之后的所有元素
>     return self.values[1:]
> def init(self):
>     # 获取最后一个元素之前的所有元素
>     return self.values[:-1]
> def last(self):
>     # 获取最后一个元素
>     return self.values[-1]
> def drop(self, n):
>     # 获取所有元素，除了前 N 个
>     return self.values[n:]
> def take(self, n):
>     # 获取前 N 个元素
>     return self.values[:n]
> ...
>

```

> 我们再举个例子，实现 Perl 语言的 **AutoVivification**，它会在你每次引用一个值未定义的属性时为你自动创建数组或者字典。

```

>
> ...
> class AutoVivification(dict):
>     """Implementation of perl's autovivification feature."""
>     def __missing__(self, key):
>         value = self[key] = type(self)()
>         return value
>
> weather = AutoVivification()
> weather['china']['guangdong']['shenzhen'] = 'sunny'
> weather['china']['hubei']['wuhan'] = 'windy'
> weather['USA']['California']['Los Angeles'] = 'sunny'
> print weather
>
> # 结果输出: {'china': {'hubei': {'wuhan': 'windy'}, 'guangdong': {'shenzhen': 'sunny'}},
'USA': {'California': {'Los Angeles': 'sunny'}}}
> ...
>

```

> 在 Python 中，关于自定义容器的实现还有更多实用的例子，但只有很少一部分能够集成在 Python 标准库中，比如 [Counter, OrderedDict 等] (<https://docs.python.org/2/library/collections.html>)


```
> [回到顶部](https://www.cnblogs.com/zhoubixian/p/11129347.html#_labelTop)
>
> ## 上下文管理
>
> `with` 声明是从 Python2.5 开始引进的关键词。你应该遇过这样子的代码：
>
> ...
> with open('foo.txt') as bar:
>     # do something with bar
> ...
>
> 在 with 声明的代码段中，我们可以做一些对象的开始操作和清除操作，还能对异常进行处理。
> 这需要实现两个魔术方法：`__enter__` 和 `__exit__`。
>
> ...
> __enter__(self)
> ...
>
> `__enter__` 会返回一个值，并赋值给 `as` 关键词之后的变量。在这里，你可以定义代码段开始的一些操作。
>
> ...
> __exit__(self, exception_type, exception_value, traceback)
> ...
>
> `__exit__` 定义了代码段结束后的一些操作，可以这里执行一些清除操作，或者做一些代码段结束后需要立即执行的命令，比如文件的关闭，socket 断开等。如果代码段成功结束，那么 exception_type, exception_value, traceback 三个参数传进来时都将为 None。如果代码段抛出异常，那么传进来的三个参数将分别为：异常的类型，异常的值，异常的追踪栈。
> 如果 `__exit__` 返回 True，那么 with 声明下的代码段的一切异常将会被屏蔽。
> 如果 `__exit__` 返回 None，那么如果有异常，异常将正常抛出，这时候 with 的作用将不会显现出来。
>
> 举例说明：
>
> 这该示例中，IndexError 始终会被隐藏，而 TypeError 始终会抛出。
>
> ...
> class DemoManager(object):
>
>     def __enter__(self):
>         pass
>
>     def __exit__(self, ex_type, ex_value, ex_tb):
>         if ex_type is IndexError:
>             print ex_value.__class__
>             return True
>         if ex_type is TypeError:
>             print ex_value.__class__
>             return # return None
>
> with DemoManager() as nothing:
>     data = [1, 2, 3]
>     data[4] # raise IndexError, 该异常被 __exit__ 处理了
>
> with DemoManager() as nothing:
>     data = [1, 2, 3]
```

```

> data['a'] # raise TypeError, 该异常没有被__exit__处理
>
> '''
> 输出:
> <type 'exceptions.IndexError'>
> <type 'exceptions.TypeError'>
> Traceback (most recent call last):
> ...
> '''
> ```
>
> [回到顶部](https://www.cnblogs.com/zhoudyixian/p/11129347.html#_labelTop)
>
> ## 对象的序列化
>
> Python 对象的序列化操作是 pickling 进行的。pickling 非常的重要,以至于 Python 对此有单独的模块`pickle`, 还有一些相关的魔术方法。使用 pickling, 你可以将数据存储在文件中, 之后又从文件中进行恢复。
>
> 下面举例来描述 pickle 的操作。从该例子中也可以看出,如果通过 pickle.load 初始化一个对象, 并不会调用`__init__`方法。
>
> ```
> # -*- coding: utf-8 -*-
> from datetime import datetime
> import pickle
>
> class Distance(object):
>
>     def __init__(self, meter):
>         print 'distance __init__'
>         self.meter = meter
>
> data = {
>     'foo': [1, 2, 3],
>     'bar': ('Hello', 'world!'),
>     'baz': True,
>     'dt': datetime(2016, 10, 01),
>     'distance': Distance(1.78),
> }
> print 'before dump:', data
> with open('data.pkl', 'wb') as jar:
>     pickle.dump(data, jar) # 将数据存储在文件中
>
> del data
> print 'data is deleted!'
>
> with open('data.pkl', 'rb') as jar:
>     data = pickle.load(jar) # 从文件中恢复数据
> print 'after load:', data
> ```
>
> 值得一提,从其他文件进行 pickle.load 操作时, 需要注意有恶意代码的可能性。另外, Python 的各个版本之间,pickle 文件可能是互不兼容的。
>
> pickling 并不是 Python 的内建类型, 它支持所有实现 pickle 协议(可理解为接口)的类。pickle 协议有

```

以下几个可选方法来自定义 Python 对象的行为。

```
>
> ...
> __getinitargs__(self)
> ...
>
> 如果你希望 unpickle 时, `__init__` 方法能够调用, 那么就需要定义 `__getinitargs__`, 该方法需要
返回一系列参数的元组, 这些参数就是传给 `__init__` 的参数。
>
> 该方法只对 `old-style class` 有效。所谓 `old-style class`, 指的是不继承自任何对象的类, 往往定义
时这样表示: `class A:`, 而非 `class A(object):`
>
> ...
> __getnewargs__(self)
> ...
>
> 跟 `__getinitargs__` 很类似, 只不过返回的参数元组将传值给 `__new__`
>
> ...
> __getstate__(self)
> ...
>
> 在调用 `pickle.dump` 时, 默认是对象的 `__dict__` 属性被存储, 如果你要修改这种行为, 可以在 `__getstate__` 方法中返回一个 state。state 将在调用 `pickle.load` 时传值给 `__setstate__`
>
> ...
> __setstate__(self, state)
> ...
>
> 一般来说, 定义了 `__getstate__`, 就需要相应地定义 `__setstate__` 来对 `__getstate__` 返回的 state
进行处理。
>
> ...
> __reduce__(self)
> ...
>
> 如果 pickle 的数据包含了自定义的扩展类 (比如使用 C 语言实现的 Python 扩展类) 时, 就需要通过实现 `__reduce__` 方法来控制行为了。由于使用过于生僻, 这里就不展开继续讲解了。
>
> 令人容易混淆的是, 我们知道, `reduce()` 是 Python 的一个内建函数, 需要指出 `__reduce__` 并非定义了 `reduce()` 的行为, 二者没有关系。
>
> ...
> __reduce_ex__(self)
> ...
>
> `__reduce_ex__` 是为了兼容性而存在的, 如果定义了 `__reduce_ex__`, 它将代替 `__reduce__` 执行。
>
> 下面的代码示例很有意思, 我们定义了一个类 Slate (中文是板岩的意思)。这个类能够记录历史上每次写入
给它的值, 但每次 `pickle.dump` 时当前值就会被清空, 仅保留了历史。
>
> ...
> # -*- coding: utf-8 -*-
> import pickle
> import time
```

```

>
>classSlate:
>    '''Class to store a string and a changelog, and forget its value when pickled.'''
>    def__init__(self, value):
>        self.value = value
>        self.last_change = time.time()
>        self.history = []
>
>    defchange(self, new_value):
>        # 修改 value, 将上次的 value 记录在 history
>        self.history.append((self.last_change, self.value))
>        self.value = new_value
>        self.last_change = time.time()
>
>    defprint_changes(self):
>        print 'Changelog for Slate object:'
>        for k, v in self.history:
>            print '%s    %s' % (k, v)
>
>    def__getstate__(self):
>        # 故意不返回 self.value 和 self.last_change,
>        # 以便每次 unpickle 时清空当前的状态, 仅仅保留 history
>        return self.history
>
>    def__setstate__(self, state):
>        self.history = state
>        self.value, self.last_change = None, None
>
> slate = Slate(0)
> time.sleep(0.5)
> slate.change(100)
> time.sleep(0.5)
> slate.change(200)
> slate.change(300)
> slate.print_changes() # 与下面的输出历史对比
> with open('slate.pkl', 'wb') as jar:
>     pickle.dump(slate, jar)
> del slate # delete it
> with open('slate.pkl', 'rb') as jar:
>     slate = pickle.load(jar)
> print 'current value:', slate.value # None
> print slate.print_changes() # 输出历史记录与上面一致
> ```
>
> [回到顶部](https://www.cnblogs.com/zhouyixian/p/11129347.html#_labelTop)
>
> ### 运算符相关的魔术方法
>
> 运算符相关的魔术方法实在太多了, 也很好理解, 不打算多讲。在其他语言里, 也有重载运算符的操作, 所以我们对这些魔术方法已经很了解了。
>
>
>
> #### 比较运算符
>
> ```

```

```

> __cmp__(self, other)
> """
>
> 如果该方法返回负数, 说明`self < other`; 返回正数, 说明`self > other`; 返回 0 说明`self == ot
her`。
> 强烈不推荐来定义`__cmp__`, 取而代之, 最好分别定义`__lt__`等方法从而实现比较功能。
> `__cmp__`在 Python3 中被废弃了。
>
> """
> __eq__(self, other)
> """
>
> 定义了比较操作符`==`的行为。
>
> """
> __ne__(self, other)
> """
>
> 定义了比较操作符`!=`的行为。
>
> """
> __lt__(self, other)
> """
>
> 定义了比较操作符`<`的行为。
>
> """
> __gt__(self, other)
> """
>
> 定义了比较操作符`>`的行为。
>
> """
> __le__(self, other)
> """
>
> 定义了比较操作符`<=`的行为。
>
> """
> __ge__(self, other)
> """
>
> 定义了比较操作符`>=`的行为。
>
> 下面我们定义一种类型 Word, 它会使用单词的长度来进行大小的比较, 而不是采用 str 的比较方式。
> 但是为了避免`Word('bar') == Word('foo')`这种违背直觉的情况出现, 并没有定义`__eq__`, 因此 W
ord 会使用它的父类(str)中的`__eq__`来进行比较。
>
> 下面的例子中也可以看出: 在编程语言中, 如果`a >= b and a <= b`, 并不能推导出`a == b`这样的结论。
>
> """
> # -*- coding: utf-8 -*-
> class Word(str):
>     '''存储单词的类, 定义比较单词的几种方法'''
>     def __new__(cls, word):

```

```

>         # 注意我们必须要用到__new__方法, 因为 str 是不可变类型
>         # 所以我们要在创建的时候将它初始化
>         if ' ' in word:
>             print "Value contains spaces. Truncating to first space."
>             word = word[:word.index(' ')] # 单词是第一个空格之前的所有字符
>         return str.__new__(cls, word)
>
>     def __gt__(self, other):
>         return len(self) > len(other)
>     def __lt__(self, other):
>         return len(self) < len(other)
>     def __ge__(self, other):
>         return len(self) >= len(other)
>     def __le__(self, other):
>         return len(self) <= len(other)
>
> print 'foo < fool:', Word('foo') < Word('fool') # True
> print 'foolish > fool:', Word('foolish') > Word('fool') # True
> print 'bar >= foo:', Word('bar') >= Word('foo') # True
> print 'bar <= foo:', Word('bar') <= Word('foo') # True
> print 'bar == foo:', Word('bar') == Word('foo') # False, 用了 str 内置的比较方法来进行比
较
> print 'bar != foo:', Word('bar') != Word('foo') # True
> ```
>
>
>
> #### 一元运算符和函数
>
> ```
> __pos__(self)
> ```
>
> 实现了 '+' 号一元运算符(比如 `+some_object`)
>
> ```
> __neg__(self)
> ```
>
> 实现了 '-' 号一元运算符(比如 `-some_object`)
>
> ```
> __invert__(self)
> ```
>
> 实现了 `~` 号(波浪号)一元运算符(比如 `~some_object`)
>
> ```
> __abs__(self)
> ```
>
> 实现了 `abs()` 内建函数.
>
> ```
> __round__(self, n)
> ```

```

```
>
> 实现了`round()`内建函数。参数 n 表示四舍五进的精度。
>
> ...
> __floor__(self)
> ...
>
> 实现了`math.floor()`，向下取整。
>
> ...
> __ceil__(self)
> ...
>
> 实现了`math.ceil()`，向上取整。
>
> ...
> __trunc__(self)
> ...
>
> 实现了`math.trunc()`，向 0 取整。
>
>
>
> ### 算术运算符
>
> ...
> __add__(self, other)
> ...
>
> 实现了加号运算。
>
> ...
> __sub__(self, other)
> ...
>
> 实现了减号运算。
>
> ...
> __mul__(self, other)
> ...
>
> 实现了乘法运算。
>
> ...
> __floordiv__(self, other)
> ...
>
> 实现了`//`运算符。
>
> ...
> __div__(self, other)
> ...
>
> 实现了`/`运算符。该方法在 Python3 中废弃。原因是 Python3 中，division 默认就是 true division。
>
> `__truediv__`(self, other)
```

```
>
> 实现了 true division. 只有你声明了`from __future__ import division`该方法才会生效.
>
> ```
> __mod__(self, other)
> ```
>
>
> 实现了`%`运算符, 取余运算.
>
> ```
> __divmod__(self, other)
> ```
>
>
> 实现了`divmod()`内建函数.
>
> ```
> __pow__(self, other)
> ```
>
>
> 实现了`**`操作. N 次方操作.
>
> ```
> __lshift__(self, other)
> ```
>
>
> 实现了位操作`<<`.
>
> ```
> __rshift__(self, other)
> ```
>
>
> 实现了位操作`>>`.
>
> ```
> __and__(self, other)
> ```
>
>
> 实现了位操作`&`.
>
> ```
> __or__(self, other)
> ```
>
>
> 实现了位操作`|`
>
> ```
> __xor__(self, other)
> ```
>
>
> 实现了位操作`^`
>
>
>
>
> ### 反算术运算符
>
```



```
> - imod (self, other)
```

```

> - `__ipow__(self, other)`
> - `__ilshift__(self, other)`
> - `__irshift__(self, other)`
> - `__iand__(self, other)`
> - `__ior__(self, other)`
> - `__ixor__(self, other)`
>
>
>
>
> ### 类型转化
>
> ...
> __int__(self)
> ...
>
> 实现了类型转化为 int 的行为.
>
> ...
> __long__(self)
> ...
>
> 实现了类型转化为 long 的行为.
>
> ...
> __float__(self)
> ...
>
> 实现了类型转化为 float 的行为.
>
> ...
> __complex__(self)
> ...
>
> 实现了类型转化为 complex(复数, 也即 1+2j 这样的虚数)的行为.
>
> ...
> __oct__(self)
> ...
>
> 实现了类型转化为八进制数的行为.
>
> ...
> __hex__(self)
> ...
>
> 实现了类型转化为十六进制数的行为.
>
> ...
> __index__(self)
> ...
>
> 在切片运算中将对象转化为 int, 因此该方法的返回值必须是 int。用一个例子来解释这个用法。
>
> ...
> class Thing(object):
>     def __index__(self):

```

```

>         return 1
>
> thing = Thing()
> list_ = ['a', 'b', 'c']
> print list_[thing] # 'b'
> print list_[thing:thing] # []
> ```
>
> 上面例子中, `list_[thing]` 的表现跟 `list_[1]` 一致, 正是因为 Thing 实现了 `__index__` 方法。
>
> 可能有的人会想, `list_[thing]` 为什么不是相当于 `list_[int(thing)]` 呢? 通过实现 Thing 的 `__int__` 方法能否达到这个目的呢?
>
> 显然不能。如果真的是这样的话, 那么 `list_[1.1:2.2]` 这样的写法也应该是通过的。
> 而实际上, 该写法会抛出 TypeError: 'slice indices must be integers or None or have an __index__ method'
>
> 下面我们再做个例子, 如果对一个 dict 对象执行 `dict_[thing]` 会怎么样呢?
>
> ```
> dict_ = {1: 'apple', 2: 'banana', 3: 'cat'}
> print dict_[thing] # raise KeyError
> ```
>
> 这个时候就不是调用 `__index__` 了。虽然 `list` 和 `dict` 都实现了 `__getitem__` 方法, 但是它们的实现方式是不一样的。
> 如果希望上面例子能够正常执行, 需要实现 Thing 的 `__hash__` 和 `__eq__` 方法。
>
> ```
> class Thing(object):
>     def __hash__(self):
>         return 1
>     def __eq__(self, other):
>         return hash(self) == hash(other)
>
> dict_ = {1: 'apple', 2: 'banana', 3: 'cat'}
> print dict_[thing] # apple
> __coerce__(self, other)
> ```
>
> 实现了混合模式运算。
>
> 要了解这个方法, 需要先了解 `coerce()` 内建函数: [官方文档](https://docs.python.org/2/library/functions.html#coerce) 上的解释是, coerce(x, y) 返回一组数字类型的参数, 它们被转化为同一种类型, 以便它们可以使用相同的算术运算符进行操作。如果过程中转化失败, 抛出 TypeError。
>
> 比如对于 `coerce(10, 10.1)`, 因为 10 和 10.1 在进行算术运算时, 会先将 10 转为 10.0 再来运算。因此 `coerce(10, 10.1)` 返回值是 (10.0, 10.1)。
>
> `__coerce__` 在 Python3 中废弃了。
>
> [回到顶部](https://www.cnblogs.com/zhouyixian/p/11129347.html#_labelTop)
>
> ## 其他魔术方法
>

```

```

> 还没讲到的魔术方法还有很多，但有些我觉得很简单，或者很少见，就不再累赘展开说明了。
>
> ...
> __str__(self)
> ...
>
> 对实例使用`str()`时调用。
>
> ...
> __repr__(self)
> ...
>
> 对实例使用`repr()`时调用。`str()`和`repr()`都是返回一个代表该实例的字符串，
> 主要区别在于：`str()`的返回值要方便人来看，而`repr()`的返回值要方便计算机看。
>
> ...
> __unicode__(self)
> ...
>
> 对实例使用`unicode()`时调用。`unicode()`与`str()`的区别在于：前者返回值是`unicode`，后者返回值
是`str`。`unicode`和`str`都是`basestring`的子类。
>
> 当你对一个类只定义了`__str__`但没定义`__unicode__`时，`__unicode__`会根据`__str__`的返回值
自动实现，即`return unicode(self.__str__())`；
> 但返回来则不成立。
>
> ...
> class StrDemo2:
>     def __str__(self):
>         return 'StrDemo2'
>
> class StrDemo3:
>     def __unicode__(self):
>         return u'StrDemo3'
>
> demo2 = StrDemo2()
> print str(demo2) # StrDemo2
> print unicode(demo2) # StrDemo2
>
> demo3 = StrDemo3()
> print str(demo3) # <__main__.StrDemo3 instance>
> print unicode(demo3) # StrDemo3
> __format__(self, formatstr)
> ...
>
> `"Hello, {0:abc}".format(a)`等价于`format(a, "abc")`，等价于`a.__format__("abc")`。
>
> 这在需要格式化展示对象的时候非常有用，比如格式化时间对象。
>
> ...
> __hash__(self)
> ...
>
> 对实例使用`hash()`时调用，返回值是数值类型。
>

```

```

> ...
> __nonzero__(self)
> ...
>
> 对实例使用`bool()`时调用，返回 True 或者 False。
> 你可能会问，为什么不是命名为`__bool__`？我也不知道。
> 我只知道该方法在 Python3 中改名为`__bool__`了。
> ...
> __dir__(self)
> ...
>
> 对实例使用`dir()`时调用。通常实现该方法是没必要的。
> ...
> __sizeof__(self)
> ...
>
> 对实例使用`sys.getsizeof()`时调用。返回对象的大小，单位是 bytes。
> ...
> __instancecheck__(self, instance)
> ...
>
> 对实例调用`isinstance(instance, class)`时调用。返回值是布尔值。它会判断 instance 是否是该类的实例。
> ...
> __subclasscheck__(self, subclass)
> ...
>
> 对实例使用`issubclass(subclass, class)`时调用。返回值是布尔值。它会判断 subclass 否是该类的子类。
> ...
> __copy__(self)
> ...
>
> 对实例使用`copy.copy()`时调用。返回"浅复制"的对象。
> ...
> __deepcopy__(self, memodict={})
> ...
>
> 对实例使用`copy.deepcopy()`时调用。返回"深复制"的对象。
> ...
> __call__(self, [args...])
> ...
>
> 该方法允许类的实例跟函数一样表现：
> ...
> class XClass:
>     def __call__(self, a, b):

```

```

>         return a + b
>
> def add(a, b):
>     return a + b
>
> x = XClass()
> print 'x(1, 2)', x(1, 2)
> print 'callable(x)', callable(x) # True
> print 'add(1, 2)', add(1, 2)
> print 'callable(add)', callable(add) # True
> ```
>
> [回到顶部](https://www.cnblogs.com/zhoudiyixian/p/11129347.html#_labelTop)
>
> ## Python3 中的差异
>
> - Python3 中, str 与 unicode 的区别被废除了,因而`__unicode__`没有了,取而代之地出现了`__bytes__`.
>
> - Python3 中, division 默认就是 true division, 因而`__div__`废弃.
>
> - `__coerce__`因存在冗余而废弃.
>
> - `__cmp__`因存在冗余而废弃.
>
> - `__nonzero__`改名为`__bool__`.
>
> 为中华之崛起而读书。 --孙中山
>
> # Python 魔法方法
>
> 在 python 中,有一些内置好的特定的方法,这些方法在进行特定的操作时会自动被调用,称之为魔法方法,下面介绍几种常见的魔法方法。
>
> > 1、**`__init__`**: 初始化函数,在创建实例对象为其赋值时使用,在`__new__`之后,`__init__`必须至少有一个参数 self,就是这个`__new__`返回的实例,`__init__`是在`__new__`的基础上可以完成一些其它初始化的动作,`__init__`不需要返回值。
>
> > ![img](https://upload-images.jianshu.io/upload_images/6857741-e2c4848d1b98f8aa.png?imageMogr2/auto-orient/strip|imageView2/2/w/347/format/webp)
>
> > 2、**`__new__`**: 很多人认为`__init__`是类的构造函数,其实不太确切,`__init__`更多的是负责初始化操作,相当于一个项目中的配置文件,`__new__`才是真正的构造函数,创建并返回一个实例对象,如果`__new__`只调用了一次,就会得到一个对象。**继承自 object 的新式类才有`__new__`这一魔法方法**,`__new__`至少必须要有一个参数 cls,代表要实例化的类,此参数在实例化时由 Python 解释器自动提供,**`__new__`必须要有返回值,返回实例化出来的实例(很重要)**,这点在自己实现`__new__`时要特别注意,可以 return 父类`__new__`出来的实例,或者直接是 object 的`__new__`出来的实例,若`__new__`没有正确返回**当前类 cls**的实例,那`__init__`是会被调用的,即使是父类的实例也不行。`__new__`是唯一在实例创建之前执行的方法,一般用在定义元类时使用。
>
> > 创建对象的步骤:
>
> > a、首先调用`__new__`得到一个对象
>
> > b、调用`__init__`为对象添加属性
>
> > c、将对象赋值给变量
>

```

```
> 下面来看一个结合__init__和__new__两个魔法方法的例子：
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-6d18fb23629d84cc.png?imageMogr2/auto-orient/strip|imageView2/2/w/755/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-31eb4c77603c629c.png?imageMogr2/auto-orient/strip|imageView2/2/w/372/format/webp)
>
> 从运行结果可以看出，__new__中的参数 cls 和 B 的 id 是相同的，表明__new__中默认的参数 cls 就是 B 类本身，而在 return 时，并没有正确返回当前类 cls 的实例，而是返回了其父类 A 的实例，因此__init__这一魔法方法并没有被调用，此时__new__虽然是写在 B 类中的，但其创建并返回的是一个 A 类的实例对象。
>
> 现在将 return 中的参数 A 变为 cls，再来看一下运行结果：
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-bfba42f6c5626e3a.png?imageMogr2/auto-orient/strip|imageView2/2/w/647/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-de52dc2041329e28.png?imageMogr2/auto-orient/strip|imageView2/2/w/411/format/webp)
>
> 可以看出，当__new__正确返回其当前类 cls 的实例对象时，__init__被调用了，此时创建并返回的是一个 B 类的实例对象。
>
> 3、**__class__**：获得已知对象的类（对象.__class__）。
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-f096dcc7e11df9f9.png?imageMogr2/auto-orient/strip|imageView2/2/w/369/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-0d2fbefe879b4adf.png?imageMogr2/auto-orient/strip|imageView2/2/w/322/format/webp)
>
> __class__至少在下面这种情况中是有用的：即当一个类中的某个成员变量是所有该类的对象的公共变量时，下面看一个例子：
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-52c2179e3934c4f9.png?imageMogr2/auto-orient/strip|imageView2/2/w/500/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-3b9ca1b0c853878c.png?imageMogr2/auto-orient/strip|imageView2/2/w/408/format/webp)
>
> 从运行结果可以看出，虽然 a 和 b 是两个不同的 A 类的实例对象，但采用了__class__之后，分别调用两个对象的 addcount 方法之后，获取到的对象的 count 属性却是在不断累加的，此时 self.__class__.count 不再是单纯的某个对象私有的属性，而是类的所有实例对象的共有属性，它相当于 self.A.count。若将 self.__class__.count += 1 变为 self.count += 1，此时__class__的效果就十分明显了。
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-09b3c617b2fa3e5b.png?imageMogr2/auto-orient/strip|imageView2/2/w/400/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-a1c96ab664a678b2.png?imageMogr2/auto-orient/strip|imageView2/2/w/453/format/webp)
>
> 4、**__str__**：在将对象转换成字符串 str(对象) 测试的时候，打印对象的信息，**__str__**方法必须要 return 一个字符串类型的返回值，作为对实例对象的字符串描述**，__str__实际上是被 print 函数默认调用的，当要 print（实例对象）时，默认调用__str__方法，将其字符串描述返回。如果不是要用 str()函数转换。当你打印一个类的时候，那么 print 首先调用的就是类里面的定义的__str__。
```

```
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-3e60fd4b093c7de3.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/710/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-5b8e614d6894e6c8.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/515/format/webp)
>
> 下面将上述代码放在控制台运行，
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-9346f2459ad4f703.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/587/format/webp)
>
> 可以看出，直接敲 a 的话，__str__ 方法是不会被调用的，而 print(a) 的时候，__str__ 就被调用了。
>
> 5、**__repr__**：如果说__str__体现的是一种可读性，是给用户看的，那么__repr__方法体现的则是一种准确性，是给开发人员看的，它对应的是 repr() 函数，重构__repr__方法后，在控制台直接敲出实例对象的名称，就可以按照__repr__中 return 的值显示了。
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-3b9447e371d9d89d.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/634/format/webp)
>
> 打印操作会首先尝试__str__和 str 内置函数(print 运行的内部等价形式)，它通常应该返回一个友好的显示。
>
> __repr__用于所有其他的环境中：用于交互模式下提示回应以及 repr 函数，它通常应该返回一个编码字符串，可以用来重新创建对象，或者给开发者详细的显示。
>
> 当我们想所有环境下都统一显示的话，可以重构__repr__方法；当我们想在不同环境下支持不同的显示，例如终端用户显示使用__str__，而程序员在开发期间则使用底层的__repr__来显示，实际上__str__只是覆盖了__repr__以得到更友好的用户显示。
>
> 6、**__del__**：对象在程序运行结束之后进行垃圾回收的时候调用这个方法，来释放资源。此时，此方法是被自动调用的。除非有特殊要求，一般不要重写。在关闭数据库连接对象的时候，可以在这里，释放资源。
>
> 看一个例子：
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-ef72cac77b19c3fc.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/707/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-4eaa9f192eb5e6df.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/345/format/webp)
>
> 可以看出在程序运行结束之后，__del__默认被调用了三次，分别对实例对象 aa,bb,cc 进行垃圾回收，因为此时创建的实例已经没有对象再指向它了。下面再看一个例子：
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-ba714a2b90b1bb85.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/587/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-83cba0b5d9342f7c.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/553/format/webp)
>
> 可以看出，wangcai 和 xiaoqiang 指向的是同一个实例对象，在 del wangcai 的时候，__del__并没有被调用，因为此时这个对象还在被 xiaoqiang 引用着，当 del xiaoqiang 的时候，__del__就默认被调用了，因为此时没有变量再引用这个实例对象了，相当于其引用计数变为 0 了，这个对象理所当然就会被垃圾回收。
```


>
> 总而言之，`__del__`魔法方法是在对象没有变量再引用，其引用计数减为 0，进行垃圾回收的时候自动调用的。
>
> 7、**`__getattribute__`**: 属性访问拦截器，在访问实例属性时自动调用。在 python 中，类的属性和方法都理解为属性，且均可以通过`__getattribute__`获取。当获取属性时，相当于对属性进行重写，直接 `return object.__getattribute__(self, *args, **kwargs)` 或者根据判断 `return` 所需要的重写值，如果需要获取某个方法的返回值时，则需要在函数后面加上一个 `()` 即可。如果不加的话，返回的是函数引用地址。下面看一个例子：
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-dbbc5392de2bc87e.png?imageMogr2/auto-orient/strip|imageView2/2/w/617/format/webp)`
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-dd691554f95887b4.png?imageMogr2/auto-orient/strip|imageView2/2/w/503/format/webp)`
>
> 在创建实例对象 `s` 并对其初始化的时候，`subject1` 的值设置为 'python'，`subject2` 的值设置为 'cpp'，在访问 `s` 的 `subject1` 属性时，因为 `Test` 类对 `object` 类中的 `__getattribute__` 方法进行了重写，所以在调用此方法时，首先对要访问的属性做一个拦截和判断，此时 `__getattribute__` 方法中的参数 `obj` 对应的是要访问的属性，若要访问 `subject1` 属性，则对该属性进行重写，返回了一个不同的字符串，我们可以看到，在初始化时，`subject1` 的值为 'python'，而在访问 `subject1` 这个属性时，返回的值是 'redirect python'，而在访问 `subject2` 时，则调用其父类中的 `__getattribute__` 方法，返回正常的 `subject2` 属性的值。当然，在访问类的方法属性时，也可以通过重写 `__getattribute__` 的方法对其进行重写。
>
> 8、**`__bases__`**: 获取指定类的所有父类构成元素，使用方法为类名.`__bases__`
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-5cecf9742bbef107.png?imageMogr2/auto-orient/strip|imageView2/2/w/639/format/webp)`
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-dadb1aff9e25931f.png?imageMogr2/auto-orient/strip|imageView2/2/w/512/format/webp)`
>
> 9、**`__mro__`**: 显示指定类的所有继承脉络和继承顺序，假如这个指定的类不具有某些方法和属性，但与其有血统关系的类中具有这些属性和方法，则在访问这个类本身不具有的这些方法和属性时，会按照 `__mro__` 显示出来的顺序一层一层向后查找，直到找到为止。
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-bf844fc650ef13ac.png?imageMogr2/auto-orient/strip|imageView2/2/w/621/format/webp)`
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-5ceac82d8d02018f.png?imageMogr2/auto-orient/strip|imageView2/2/w/659/format/webp)`
>
> 10、**`__call__`**: 具有 `__call__` 魔法方法的对象可以使用 `xxx()` 的形式被调用，比如说累的实例对象
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-9763febee55784e3.png?imageMogr2/auto-orient/strip|imageView2/2/w/556/format/webp)`
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-db11f5088799c21f.png?imageMogr2/auto-orient/strip|imageView2/2/w/560/format/webp)`
>
> 可以看到，`Dog` 类的实例对象 `laowang` 是不可以使用 `laowang()` 的方式进行调用的，因为其没有 `__call__` 魔法方法，进行了修改之后，`laowang` 这个实例对象就可以使用 `()` 的方式被调用了：
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-5b181a4e2261055c.png?imageMogr2/auto-orient/strip|imageView2/2/w/578/format/webp)`
>
> `![img](https://upload-images.jianshu.io/upload_images/6857741-279dd48f7c2814b5.png?ima`

```

geMogr2/auto-orient/strip|imageView2/2/w/587/format/webp)
>
> 11、**魔法属性: **__slots__:可以限制实例对象的属性和方法, 但是对类不起作用。
>
> 12、**__all__:**将一个 py 文件作为模块导入时, 其中 if __name__ == "main"以上的类、方法、函数
等都能被导入, 但某些方法可能只是用来做测试用的, 不希望也不建议被导入, 可以用__all__=['函数名或方法
名']的方式限制一下哪些函数或方法可以被导入, 即[]中的函数名或方法名可以被导入。但是需要强调的是, **__
all__魔法方法只针对通过 from xx import \*这种导入方式有效**。
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-b0825fb96b8ffe13.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/521/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-a7e736cfb210efa6.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/700/format/webp)
>
> ![img](https://upload-images.jianshu.io/upload_images/6857741-e6d9e8a7bf2d7032.png?ima
geMogr2/auto-orient/strip|imageView2/2/w/642/format/webp)

```

4. 生成式、生成器、迭代器的编写。

生成式: 一次性生成所有数据, 然后保存在内存中, 适合小量数据

生成器: 返回一个可迭代对象, 生成器 generator 对象, 必须通过遍历才能一一将值取出

生成器不会一次列出所有的数据, next()一次, 给一个值, 更加节省内存使用率

可迭代对象 (iterable): 可以通过 for 循环调用出来的, 就是可迭代对象, 如: 列表、元组、字典、生成式、生成器

迭代器 (iterator): 任何具有 next()方法的对象都是迭代器, 对迭代器调用 next()方法可以获取下一个值

列表生成式

列表生成式即 List Comprehensions, 是 Python 内置的非常简单却强大的可以用来创建 list 的生成式。

```

...
在列表中存放 1-10 的数据

'''
#第一种方式
list0 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#第二种方式
list0 = list(range(1, 11))
#第三种方式
list0 = []
for i in range(1, 11):
    if i % 2 != 0:
        list0.append(i)
#列表中存放 1-10 的偶数, 用列表生成式来写
list0 = [i for i in range(1, 11) if i % 2 != 0]
'''

```

列表生成器的格式:

[存放于列表中的元素 元素来源 元素筛选条件]

练习:

```
list0 = [" ", "\nabc", "\t", "hello"]
```

在 list0 的基础上 快速的生成一个列表，列表的存放的元素是非空白序列

#如何判断一个序列是空白字符序列 ---> 按照空白进行切割 列表的长度为 0

去除字符序列两端空白 strip() ---> 按照空白区去除两端的内容 如果最终剩下的是空序列

```
res = [item for item in list0 if len(item.split()) != 0]
print(res)
res = [item for item in list0 if len(item.strip()) != 0]
print(res)
```

关于 strip()

```
'''
```

any whitespace string is a separator and empty strings are removed from the result.

如果使用的是空白分割符的话 会将生成的列表中的空字符序列给移除掉

```
'''
```

#判断数据是否是指定类型 isinstance()

```
print(type(a))
res = isinstance(a, str)
print(res)
```

字典生成式

```
'''
```

```
dict0 = {"英语":66, "数学":78, "政治":42, "语文":59}
```

去除掉字典中 value < 60 的数据

```
'''
```

```
dict0 = {"英语":66, "数学":78, "政治":42, "语文":59}
```

#第一种方式

```
new_dict = {}
for k, v in dict0.items():
    if v >= 60:
        new_dict[k] = v
```

#利用字典生成式

```
dict1 = {k:v for k, v in dict0.items() if v >= 60}
print(dict1)
```

生成器(generator)

Python 中，这种一边循环一边计算的机制，称为生成器：generator。

生成器可以生成一批数据，但是这些数据不会立即存放于内存中，内存中存放的是生成器对象，保存生成数据的方法。

```
'''
```

声明生成器

```
'''
```

#第一种方式

```
gene = (i for i in range(1, 10001))
```

#第二种方式 函数的 return 改为 yield

```
def test():
    for i in range(1, 10001):
        yield i
        print(i, "yield 之后执行的语句") # 当下一次获取时候回执行
```

```
...
```

获取数据

```
...
```

```
#通过 next 方法，可以获取数据，直到抛出 StopIteration 错误
print(gene.next())
print(gene.next())
...
print(gene.next())
##### StopIteration 当生成器中数据已经取完了,再获取的话就会报错
```

```
#通过 for 循环获取数据 可以忽略 StopIteration 错误
ge = test()
for i in range(10):
    value = next(ge)
    print(value)
迭代器(Iterator)
```

```
#可以使用迭代器进行遍历
from collections import Iterator
```

```
#符合这个类型的都可以通过 for in 进行遍历
from collections import Iterable
```

```
res = isinstance(11, Iterable) #False
```

```
res = isinstance("11", Iterable) #True
```

```
res = isinstance([], Iterable) #True
```

```
res = isinstance((), Iterable) #True
```

```
res = isinstance({}, Iterable) #True
```

```
res = isinstance(set(), Iterable) #True
```

能够使用 `for in` 遍历的数据不一定可以使用迭代器来进行遍历

迭代器的遍历是通过 `next` 方法进行遍历的

能使用迭代器来进行遍历 前提必须是属于 `Iterator` 这种类型

为什么 `list`、`dict`、`str` 等数据类型不是 `Iterator`?

因为 Python 的 `Iterator` 对象表示的是一个数据流，`Iterator` 对象可以被 `next()` 函数调用并不断返回下一个数据，直到没有数据时抛出 `StopIteration` 错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过 `next()` 函数实现按需计算下一个数据，所以 `Iterator` 的计算是惰性的，只有在需要返回下一个数据时它才会计算。

`Iterator` 甚至可以表示一个无限大的数据流，例如全体自然数。而使用 `list` 是永远不可能存储全体自然数的。

```
a = "11"
```

```
next(a)
print(res) # 'str' object is not an iterator
```

```
res = isinstance("", Iterator) #False
```

```
res = isinstance([], Iterator) #False
```

```
res = isinstance({}, Iterator) #False
```

```
res = isinstance((), Iterator) #False
```

```
res = isinstance(set(), Iterator) #False
```

如何将可以使用 `for in` 进行遍历的数据 也可以使用迭代器进行遍历??

可以将数据通过 `iter()` 的方法将其转化为 `Iterator` 的类型

```
ite = iter(["12", "17", "34", "99"])
```

```
value = next(ite)
```

```
print(value)
```

小结:

凡是可作用于 `for` 循环的对象都是 `Iterable` 类型;

凡是可作用于 `next()` 函数的对象都是 `Iterator` 类型, 它们表示一个惰性计算的序列;

集合数据类型如 `list`、`dict`、`str` 等是 `Iterable` 但不是 `Iterator`, 不过可以通过 `iter()` 函数获得一个 `Iterator` 对象。

1. 什么是迭代器

迭代指的是一个重复的过程, 每一次重复都是基于上一次的结果而来的

迭代器指的是迭代取值的工具, 该工具的特点是可以不依赖于索引取值

#原始的迭代方法

```
li=['a','b','c','d','e']
```

```
li=('a','b','c','d','e')
```

```
li='hello'
```

```
i=0
```

```
while i < len(li):
```

```
    print(li[i])
```

```
    i+=1
```

1. 为何要用迭代器

为了找出一种通用的&可以不依赖于索引的迭代取值方式

2. 如何用迭代器

可迭代的对象: 但凡内置有 **iter** 方法的对象都称之为可迭代的对象

迭代器对象: 既内置有 **iter** 方法, 又内置有 **next** 方法

关于 **iter** 方法:

调用可迭代对象的 **iter** 会得到一个迭代器对象

调用迭代器对象的 **iter** 会得到迭代器本身

3. 总结迭代器的优缺点:

优点:

1. 提供了一种通用的&可以不依赖于索引的迭代取值方式
2. 同一时刻在内存中只有一个值,更加节省内存(老母鸡, Python3 的 `range()`方法就相当于一只老母鸡)

缺点:

1. 取指定值不如索引灵活,并且迭代器是一次性的

4. 无法预知迭代器数据的个数

迭代器对象:

可迭代的对象: `str, list, tuple, dict, set`, 文件对象(除了数字类型)

迭代器对象: 文件对象

可迭代的对象=====》迭代器对象: 调用可迭代对象内置的 **`iter`** 方法会有一个返回值, 该返回值就是对应的迭代器对象

```
#基本语法 dic={'x':1,'y':2,'z':3}
iterdic=dic.iter()
print(iterdic) #<dictkeyiterator object at 0x0000000001D97728>
res1=iterdic.next()
print(res1) #x
res2=iterdic.next()
print(res2)
res3=iterdic.next()
print(res3)
#超出迭代器数据个数,报错
res4=iter_dic.next()
print(res4)
print(dic.iter().next())#x
print(dic.iter().next())#x
print(dic.iter().next())#x
```

通过循环来捕捉异常,避免取值越界后报错:

```
1 dic={'x':1,'y':2,'z':3}
2 # dic=['a','b','c']
3 iter_dic=dic.__iter__()
4
5 iter_dic=open(r'今日内容',mode='rt',encoding='utf-8')#文件为迭代器对象
6
7 while True:
8     try:
9         print(iter_dic.__next__())
```

```
10 except StopIteration:
```

```
11 break
```

```
# for 循环原理:
```

for 准确地说应该是迭代器循环,for 循环的原理如下:

1. 先调用 in 后面那个值的 **iter** 方法, 得到迭代器对象
. 执行迭代器.**next()**方法得到一个返回值, 然后赋值给一个变量 **k**, 运行循环体代码
3. 循环往复, 直到迭代器取值完毕抛出异常然后捕捉异常自动结束循环

```
1 dic={'x':1,'y':2,'z':3}
2 iter_dic=dic.__iter__()
3 print(iter_dic)
4 print(iter_dic.__iter__())
5
6 for k in dic: # iter_dic=dic.__iter__() | k=iter_dic.__next__()
7     print(k)
8 # <dict_keyiterator object at 0x0000000000477728>
9 # <dict_keyiterator object at 0x0000000000477728>

10 # x
11 # y
12 # z
with open(r'今日内容',mode='rt',encoding='utf-8') as f:
for line in f: #iter_f=f.iter()
print(line)
```

```
# **自定义迭代器:**
```

yield 关键字: 只能用在函数内

在函数内但凡包含有 **yield** 关键字, 再去执行函数, 就不会立刻运行函数体代码了, 会得到一个返回值, 该返回值称之为生成器对象, 生成器本质就是迭代器

总结 yield:

1. 提供一种自定义迭代器的解决方案
. yield 可用于返回值

yield VS return

相同点: 都可以用于返回值

不同点: **yield** 可以暂停函数, **yield** 可以返回多次值, 而 **return** 只能返回值一次值函数就立刻终止

```
1 def func():
2     print('=====>第一次')
3     yield 1
4     print('=====>第二次')
5     yield 2
6     print('=====>第三次')
7     yield 3
```

```

8     print('=====>第四次')
9
10 g=func()
11 print(g.iter().iter().iter() is g)
12 iter(g) #g.iter()
13 res=next(g) #g.next()
14 print(res)
15
16 res1=next(g)
17 print(res1)
18
19 res2=next(g)
20 print(res2)
21
22 res3=next(g) #报错,取不到
23 print(res3)
24 # True
25 # =====>第一次
26 # 1
27 # =====>第二次
28 # 2
29 # =====>第三次
30 # 3
31 # =====>第四次

```

自定义一个 rang 函数：

```

def myrange(start,stop,step=1):
    while start < stop:
        yield start
        start+=step
res=myrange(1,5,step=2) # 1 3
print(next(res)) #1
print(next(res)) #3
for item in res: #还在同一个迭代器中,此时 res 已经到 3,步长为 2,没有下一个值
    print(item) #打印不到结果
for item in my_range(1,5,2): #这是一个新的迭代器
    print(item) #1,3

```

三元表达式：

原始操作：

```

def max2(x,y):
    if x > y:
        return x
    else:
        return y

```


简化操作:

```
x=10
y=20
res=x if x > y else y
print(res)
```

XXX 生成式:

列表生成式:列表

将大于 4 的数字添加到 egg 后,存入列表:

#原始操作:

```
l=[]
for i in range(1,11):
    if i > 4:
        res='egg%s' %i
        l.append(res)
print(l)
#简化操作:
l=['egg%s' %i for i in range(1,11) if i > 4]
print(l)#[ 'egg5', 'egg6', 'egg7', 'egg8', 'egg9', 'egg10']
```

将除了 egon 外的名字后加上_DSB,然后存入一个列表:

#原始操作:

```
names=['egon','lxx','yyx','cw','alex','wxx']
l=[]
for name in names:
    if name != 'egon':
        res='%sDSB' %name
        l.append(res)
print(l)
#简化操作:
l=['%sDSB' %name for name in names if name != 'egon']
print(l)#[ 'lxxDSB', 'yyxDSB', 'cwDSB', 'alexDSB', 'wxx_DSB']
```

字典生成式:字典

```
items=[('name','egon'),('age',18),('sex','male')]
```

#原始方法:

```
dic={}
for k,v in items:
    dic[k]=v
print(dic)
```

#简化方法:

```
res={k:v for k,v in items if k != 'sex'}
print(res)#{'name': 'egon', 'age': 18}
res={i for i in 'hello'}
print(res)#{'o', 'h', 'e', 'l'} 去重,逐个,无序
```

生成器表达式(实际也是一个迭代器):

```
res=(i**2 for i in range(3))
print(res) #本身是一个迭代器,只有调用 next 方法,才能取到值,什么也不会打印
print(next(res))
print(next(res))
print(next(res))
#0
#1
#4
```

```
1 with open(r'今日内容',mode='rt',encoding='utf-8') as f:
2     data=f.read()
3     print(len(data)) #1025
4
5 with open(r'今日内容', mode='rt', encoding='utf-8') as f:
6     res=0
7     for line in f:
8         res+=len(line)
9     print(res)#1025
10
11 with open(r'今日内容',mode='rt',encoding='utf-8') as f:
12 res=sum([len(line) for line in f])
13 # res=sum(len(line) for line in f)#有双重小括号,可以省略一个
14 print(res)
15
16 with open(r'今日内容',mode='rt', encoding='utf-8') as f:
17 res=max([len(line) for line in f]) #60
18 #res=max([len(line) for line in f]) #60
19 #res=max(len(line) for line in f) #60
20 print(res)
```

生成器的 `send()` 和 `close()` 方法

生成器中还有两个很重要的方法: `send()` 和 `close()`。

- `send(value)`:

从前面了解到, `next()` 方法可以恢复生成器状态并继续执行, 其实 `send()` 是除 `next()` 外另一个恢复生成器的方法。

Python 2.5 中, `yield` 语句变成了 `yield` 表达式, 也就是说 `yield` 可以有一个值, 而这个值就是 `send()` 方法的参数, 所以 `send(None)` 和 `next()` 是等效的。同样, `next()` 和 `send()` 的返回值都是 `yield` 语句处的参数 (`yielded value`)

关于 `send()` 方法需要**注意**的是: 调用 `send` 传入非 `None` 值前, 生成器必须处于挂起状态, 否则将抛出异常。也就是说, 第一次调用时, 要使用 `next()` 语句或 `send(None)`, 因为没有 `yield` 语句来接收这个值。

- `close()`:

这个方法用于关闭生成器，对关闭的生成器后再次调用 `next` 或 `send` 将抛出 `StopIteration` 异常。

总结

本文介绍了 Python 迭代器和生成器的相关内容。

- 通过实现迭代器协议对应的 `__iter__()` 和 `next()` 方法，可以自定义迭代器类型。对于可迭代对象，`for` 语句可以通过 `iter()` 方法获取迭代器，并且通过 `next()` 方法获得容器的下一个元素。
- 像列表这种序列类型的对象，可迭代对象和迭代器对象是相互独立存在的，在迭代的过程中各个迭代器相互独立；但是，有的可迭代对象本身又是迭代器对象，那么迭代器就没法独立使用。
- `itertools` 模块提供了一系列迭代器，能够帮助用户轻松地使用排列、组合、笛卡尔积或其他组合结构。
- 生成器是一种特殊的迭代器，内部支持了生成器协议，不需要明确定义 `__iter__()` 和 `next()` 方法。
- 生成器通过生成器函数产生，生成器函数可以通过常规的 `def` 语句来定义，但是不用 `return` 返回，而是用 `yield` 一次返回一个结果。

5. 列表、集合、字典的底层实现。

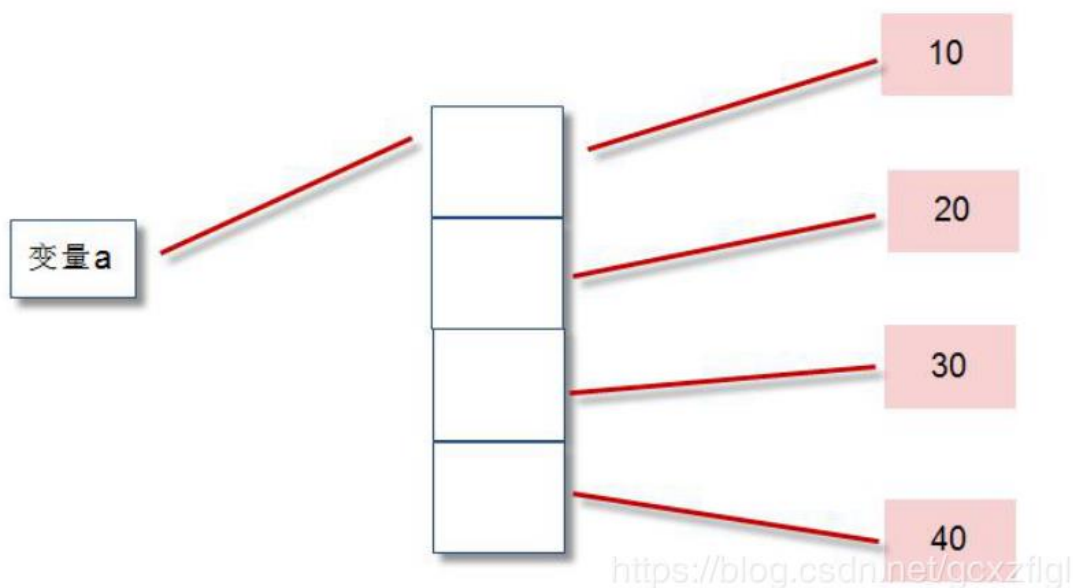
Python 入门序列（列表，元组，字典，集合）

序列

序列是一种数据存储方式，用来存储一系列的数据。在内存中，序列就是一块用来存放多个值的连续的内存空间。比如一个整数序列 `[10,20,30,40]`，可以这样示意表示：

由于 Python3 中一切皆对象，在内存中实际是按照如下方式存储的：

`a = [10,20,30,40]`



从图示中，我们可以看出序列中存储的是整数对象的地址，而不是整数对象的值。python 中常用的序列结构有：字符串、列表、元组、字典、集合

列表简介

列表：用于存储任意数目、任意类型的数据集合。

列表是内置可变序列，是包含多个元素的有序连续的内存空间。列表定义的标准语法格式：

a = [10,20,30,40]其中，10,20,30,40 这些称为：列表 a 的元素。

列表中的元素可以各不相同，可以是任意类型。比如：a = [10,20,'abc',True]

方法	要点	描述
list.append(x)	增加元素	将元素 x 增加到列表 list 尾部
list.extend(aList)	增加元素	将列表 alist 所有元素加到列表 list 尾部
list.insert(index,x)	增加元素	在列表 list 指定位置 index 处插入元素 x
list.remove(x)	删除元素	在列表 list 中删除首次出现的指定元素 x
list.pop([index])	删除元素	删除并返回列表 list 指定为止 index 处的元素 ,默认是最后一个元素
list.clear()	删除所有元素	删除列表所有元素，并不是删除列表对象
list.index(x)	访问元素	返回第一个 x 的索引位置 若不存在 x 元素抛出异常
list.count(x)	计数	返回指定元素 x 在列表 list 中出现的次数
len(list)	列表长度	返回列表中包含元素的个数
list.reverse()	翻转列表	所有元素原地翻转
list.sort()	排序	所有元素原地排序
list.copy()	浅拷贝	返回列表对象的浅拷贝

Python 的列表大小可变，根据需要随时增加或缩小。

字符串和列表都是序列类型，一个字符串是一个字符序列，一个列表是任何元素的序列

基本语法[]创建

```
harm_workspace E:\pycharm_workspace 1 a = [10,20,True,'gcx']
env library root 2 print(a)
my01.py
my01 x my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
[10, 20, True, 'gcx']
Process finished with exit code 0 https://blog.csdn.net/gcxzflgl
```

list()创建 range()创建

使用 list()可以将任何可迭代的数据转化成列表。range()可以帮助我们非常方便的创建整数列表，语法格式为：

range([start,] end [,step])

```
harm_workspace E:\pycharm_workspace 1 a = list(range(10))
env library root 2 print(a)
my01.py 3 print(list(range(0,20,2)))
my02.py
my03.py
my04.py
...
my01 x my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18] https://blog.csdn.net/gcxzflgl
```

append()方法

原地修改列表对象，是真正的列表尾部添加新的元素，速度最快，推荐使用。

```
charm_workspace E:\pycharm_workspace 1 a = list(range(10))
venv library root 2 print(a)
my01.py 3 a.append(100)
my02.py 4 print(a)
my03.py
my01 x my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 100]
Process finished with exit code 0
```

<https://blog.csdn.net/gcxzflgl>

+运算符操作

并不是真正的尾部添加元素，而是创建新的列表对象；将原列表的元素和新列表的元素依次复制到新的列表对象中。这样，会涉及大量的复制操作，对于操作大量元素不建议使用。

```
charm_workspace E:\pycharm_workspace 1 a = list(range(10))
venv library root 2 print(id(a))
my01.py 3 a = a + [50]
my02.py 4 print(id(a))
my03.py
my01 x my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
5464872
2701872
Process finished with exit code 0
```

<https://blog.csdn.net/gcxzflgl>

通过如上测试，我们发现变量 **a** 的地址发生了变化。也就是创建了新的列表对象。

extend()方法

将目标列表的所有元素添加到本列表的尾部，属于原地操作，不创建新的列表对象。

```
charm_workspace E:\pycharm_workspace 1 a = list(range(10))
venv library root 2 print(id(a))
my01.py 3 a.extend([100])
my02.py 4 print(id(a))
my03.py
my01 x my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
13460384
13460384
Process finished with exit code 0
```

<https://blog.csdn.net/gcxzflgl>

insert()插入元素

使用 **insert()** 方法可以将指定的元素插入到列表对象的任意制定位置。这样会让插入位置后面所有的元素进行移动，会影响处理速度。涉及大量元素时，尽量避免使用。类似发生这种移动的函数还有：**remove()**、**pop()**、**del()**，它们在删除非尾部元素时也会发生操作位置后面元素的移动。

```
charm_workspace E:\pycharm_workspace 1 a = list(range(10))
venv library root 2 print(id(a))
my01.py 3 a.insert(3,100)
my02.py 4 print(a)
my03.py
my01 x my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
13657072
[0, 1, 2, 100, 3, 4, 5, 6, 7, 8, 9]
Process finished with exit code 0
```

<https://blog.csdn.net/gcxzflgl>

del 删除

删除列表指定位置的元素。



```
pycharm_workspace E:\pycharm_workspace
venv library root
my01.py
my02.py
my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
[100, 888, 300, 400]
Process finished with exit code 0
```

<https://blog.csdn.net/gcxzflgl>

pop()方法

pop()删除并返回指定位置元素，如果未指定位置则默认操作列表最后一个元素。

```
pycharm_workspace E:\pycharm_workspace
venv library root
my01.py
my02.py
my03.py
my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
[100, 200, 888, 300]
[100, 200, 300]
Process finished with exit code 0
```

<https://blog.csdn.net/gcxzflgl>

remove()方法

删除首次出现的指定元素，若不存在该元素抛出异常。

```
pycharm_workspace E:\pycharm_workspace
venv library root
my01.py
my02.py
my03.py
my04.py
my01 x
E:\pycharm_workspace\venv\Scripts\python.exe E:\pycharm_workspace\my01.py
[100, 888, 300, 400]
Traceback (most recent call last):
  File "E:\pycharm_workspace\my01.py", line 4, in <module>
    a.remove(1000)
ValueError: list.remove(x): x not in list
Process finished with exit code 1
```

<https://blog.csdn.net/gcxzflgl>

通过索引直接访问元素

可以通过索引直接访问元素。索引的区间在[0, 列表长度-1]这个范围。超过这个范围则会抛出异常。

index()获得指定元素在列表中首次出现的索引

index()可以获取指定元素首次出现的索引位置。语法是：index(value,[start,[end]])。其中，start 和 end 指定了搜索的范围。

count()获得指定元素在列表中出现的次数

count()可以返回指定元素在列表中出现的次数。

成员资格判断

判断列表中是否存在指定的元素，我们可以使用 count()方法，返回 0 则表示不存在，返回大于 0 则表示存在。但是，一般我们会使用更加简洁的 in 关键字来判断，直接返回 True 或 False。

切片操作

在前面字符串章节，提到过字符串的切片操作，对于列表的切片操作和字符串类似。

切片是 Python 序列及其重要的操作，适用于列表、元组、字符串等等。切片的格式如下：

切片 slice 操作可以让我们快速提取子列表或修改。标准格式为：[起始偏移量 start:终止偏移量 end[:步长 step]]

注：当步长省略时顺便可以省略第二个冒号

切片操作时，起始偏移量和终止偏移量不在[0,字符串长度-1]这个范围，也不会报错。起始偏移量小于 0 则会当做 0，终止偏移量大于“长度-1”会被当成“长度-1”

列表排序

修改原列表，不建新列表的排序

建新列表的排序

我们也可以通过内置函数 `sorted()` 进行排序，这个方法返回新列表，不对原列表做修改。

`reversed()` 返回迭代器

内置函数 `reversed()` 也支持进行逆序排列，与列表对象 `reverse()` 方法不同的是，内置函数 `reversed()` 不对原列表做任何修改，只是返回一个逆序排列的迭代器对象。

打印输出 `b` 发现提示是：`list_reverseiterator`。也就是一个迭代对象。同时，我们使用 `list(b)` 进行输出，发现只能使用一次。第一次输出了元素，第二次为空。那是因为迭代对象在第一次时已经遍历结束了，第二次不能再使用。(类似游标指针)

列表相关的其他内置函数汇总

元组 tuple

列表属于可变序列，可以任意修改列表中的元素。元组属于不可变序列，不能修改元组中的元素。因此，元组没有增加元素、修改元素、删除元素相关的方法。

元组的创建

通过 `()` 创建元组。小括号可以省略。

`a = (10,20,30)` 或者 `a = 10,20,30`

如果元组只有一个元素，则必须后面加逗号。这是因为解释器会把 `(1)` 解释为整数 1，`(1,)` 解释为元组。

通过 `tuple()` 创建元组 `tuple(可迭代的对象)`

总结：

`tuple()` 可以接收列表、字符串、其他序列类型、迭代器等生成元组。

`list()` 可以接收元组、字符串、其他序列类型、迭代器等生成列表。

元组的元素访问和计数

2. 元组的元素不能修改

列表关于排序的方法 `list.sorted()` 是修改原列表对象，元组没有该方法。如果要对元组排序，只能使用内置函数 `sorted(tupleObj)`

zip

`zip(列表 1, 列表 2, ...)` 将多个列表对应位置的元素组合成为元组，并返回这个 `zip` 对象。

元组总结

1. 元组的核心特点是：不可变序列。
2. 元组的访问和处理速度比列表快。
3. 与整数和字符串一样，元组可以作为字典的键，列表则永远不能作为字典的键使用。

字典介绍

字典是“键值对”的无序可变序列，字典中的每个元素都是一个“键值对”，包含：“键对象”和“值对象”。可以通过“键对象”实现快速获取、删除、更新对应的“值对象”。列表中我们通过“下标数字”找到对应的对象。字典中通过“键对象”找到对应的“值对象”。“键”是任意的不可变数据，比如：整数、浮点数、字符串、元组。但是：列表、字典、集合这些可变对象，不能作为“键”。并且“键”不可重复。“值”可以是任意的数据，并且可重复。一个典型的字典的定义方式：`a = {'name':'gaoqi','age':18,'job':'programmer'}`

字典的创建

1. 我们可以通过`{}`、`dict()`来创建字典对象。
2. 通过 `zip()`创建字典对象
3. 通过 `fromkeys` 创建值为空的字典

字典元素的访问

- 1.通过[键]获得“值”。若键不存在，则抛出异常。
2. 通过 `get()`方法获得“值”。推荐使用。优点是：指定键不存在，返回 `None`；也可以设定指定键不存在时默认返回的对象。推荐使用 `get()`获取“值对象”。
- 3.列出所有的键值对

字典元素添加、修改、删除

1. 给字典新增“键值对”。如果“键”已经存在，则覆盖旧的键值对；如果“键”不存在，则新增“键值对”
2. 使用 `update()`将新字典中所有键值对全部添加到旧字典对象上。如果 `key` 有重复，则直接覆盖。
3. 字典中元素的删除，可以使用 `del()`方法；或者 `clear()`删除所有键值对；`pop()`删除指定键值对，并返回对应的“值对象”
- 4.`popitem()`：随机删除和返回该键值对。字典是“无序可变序列”，因此没有第一个元素、最后一个元素的概念；`popitem` 弹出随机的项，因为字典并没有“最后的元素”或者其他有关顺序的概念。若想一个接一个地移除并处理项，这个方法就非常有效（因为不用首先获取键的列表）。

序列解包

序列解包可以用于元组、列表、字典。序列解包可以让我们方便的对多个变量赋值.序列解包用于字典时，默认是对“键”进行操作； 如果需要对键值对操作，则需要使用 `items()`；如果需要对“值”进行操作，则需要使用 `values()`

字典核心底层原理(重要)

字典对象的核心是散列表。散列表是一个稀疏数组（总是有空白元素的数组），数组的每个单元叫做 `bucket`。每个 `bucket` 有两部分：一个是键对象的引用，一个是值对象的引用。由于所有 `bucket` 结构和大小一致，我们可以通过偏移量来读取指 `bucket`。

将一个键值对放进字典的底层过程

假设字典 `a` 对象创建完后，数组长度为 7：

我要把“name”=“gaochenxi”这个键值对放到字典对象 `a` 中，第一步需要计算键“name”的散列值。Python 中通过 `hash()`来计算。

由于数组长度为 7，我们可以拿计算出的散列值的最右边 3 位数字作为偏移量，即“000”，十进制是数字 0。我们查看偏移量 0，对应的 `bucket` 是否为空。如果为空，则将键值对放进去。如果不为空，则依次取右边

3 位作为偏移量，即“010”，十进制是数字

2。再查看偏移量为 2 的 bucket 是否为空。直到找到为空的 bucket 将键值对放进去。流程图如下：

扩容

python 会根据散列表的拥挤程度扩容。“扩容”指的是:创造更大的数组，将原内容拷贝到新数组中。接近 2/3 时，数组就会扩容。

根据键查找“键值对”的底层过程

当我们调用 `a.get("name")`，就是根据键“name”查找到“键值对”，从而找到值对象“gaochenxi”。

第一步，我们仍然要计算“name”对象的散列值：

和存储的底层流程算法一致，也是依次取散列值的不同位置的数字。假设数组长度为 7，我们可以拿计算出的散列值的最右边 3 位数字作为偏移量，即“00”，十进制是数字 0。我们查看偏移量 0，对应的 bucket 是否为空。如果为空，则返回 `None`。如果不为空，则将这个 bucket 的键对象计算对应散列值，和我们的散列值进行比较，如果相等。则将对应“值对象”返回。如果不相等，则再依次取其他几位数字，重新计算偏移量。依次取完后，仍然没有找到。则返回 `None`。流程图如下：

用法总结：

1. 键必须可散列

- (1) 数字、字符串、元组，都是可散列的。
- (2) 自定义对象需要支持下面三点：
 - 1 支持 `hash()` 函数
 - 2 支持通过 `__eq__()` 方法检测相等性。
 - 3 若 `a==b` 为真，则 `hash(a)==hash(b)` 也为真。

2. 字典在内存中开销巨大，典型的空间换时间。

3. 键查询速度很快

4. 往字典里面添加新建可能导致扩容，导致散列表中键的次序变化。因此，不要在遍历字典的同时进行字典的修改。

集合

集合是无序可变元素不能重复。实际上集合底层是字典实现，集合的所有元素都是字典中的“键对象”，因此是不能重复的且唯一

集合创建和删除

3. 使用 `{}` 创建集合对象，并使用 `add()` 方法添加元素

4. 使用 `set()`，将列表、元组等可迭代对象转成集合。如果原来数据存在重复数据，则只保留一个。

5. `remove()` 删除指定元素；`clear()` 清空整个集合

集合相关操作，像数学中概念一样，Python 对集合也提供了并集、交集、差集等运算。给出示例：

推导式创建序列

推导式是从一个或者多个迭代器快速创建序列的一种方法。它可以将循环和条件判断结合，从而避免冗长的代码。推导式是典型的 Python 风格

列表推导式

列表推导式生成列表对象，语法如下：

[表达式 for item in 可迭代对象] 或者: {表达式 for item in 可迭代对象 if 条件判断}

```
[x for x in range(1,5)]
[1, 2, 3, 4]
[x2 for x in range(1,5)]
[2, 4, 6, 8]
[x2 for x in range(1,20) if x%5==0 ]
[10, 20, 30]
[forain "abcdefg"]
['a', 'b', 'c', 'd', 'e', 'f', 'g']
cells = [(row,col) forrow in range(1,10) for col in range(1,10)]#可以使用两个循环
for cell in cells:
print(cell)
```

字典推导式

字典的推导式生成字典对象，格式如下：

{keyexpression:valueexpressionfor 表达式 in 可迭代对象}类似于列表推导式，字典推导也可以增加 if 条件判断、多个 for 循环。

统计文本中字符出现的次数：

```
mytext = ' i love you, i love china, i love gao'
charcount = {c:mytext.count(c)for c in mytext}
char_count
{' ': 9, 'i': 4, 'l': 3, 'o': 5, 'v': 3, 'e': 3, 'y': 1, 'u': 1, ',': 2, 'h': 1, 'n': 1, 'c': 1, 'g': 1, 'a': 2}
```

集合推导式

集合推导式生成集合，和列表推导式的语法格式类似：

{表达式 foritemin 可迭代对象 }或者：{表达式 foritemin 可迭代对象 if 条件判断}

```
{x for x in range(1,100) if x%9==0}
{99, 36, 72, 9, 45, 81, 18, 54, 90, 27, 63}
```

生成器推导式（生成元组）

```
(x for x in range(1,100) if x%9==0)
<generator object at 0x0000000002BD3048>
```

我们发现提示的是“一个生成器对象”。显然，元组是没有推导式的。一个生成器只能运行一次。第一次迭代可以得到数据，第二次迭代发现数据已经没有了。

```
gnt = (x for x in range(1,100) if x%9==0)
for x in gnt:
print(x,end=' ')
9 18 27 36 45 54 63 72 81 90 99
for x in gnt:
print(x,end=' ')
```

[python]list, tuple, dictionary, set 的底层细节

list, tuple, dictionary, set 是 python 中 4 中常见的集合类型。在笔者之前的学习中，只是简单了学习它们 4 者的使用，现记录一下更深底层的知识。

列表和元组

列表和元组的区别是显然的：列表是动态的，其大小可以该标；而元组是不可变的，一旦创建就不能修改。

实现细节

python 中的列表的英文名是 list，因此很容易和其它语言(C++, Java 等)标准库中常见的链表混淆。事实上 CPython 的列表根本不是列表（可能换成英文理解起来容易些：python 中的 list 不是 list）。在 CPython 中，列表被实现为长度可变的数组。

从细节上看，Python 中的列表是由对其它对象的引用组成的连续数组。指向这个数组的指针及其长度被保存在一个列表头结构中。这意味着，每次添加或删除一个元素时，由引用组成的数组需要该标大小（重新分配）。幸运的是，Python 在创建这些数组时采用了指数过分配，所以并不是每次操作都需要改变数组的大小。但是，也因为这个原因添加或取出元素的平摊复杂度较低。

不幸的是，在普通链表上“代价很小”的其它一些操作在 Python 中计算复杂度相对过高。

利用 `list.insert` 方法在任意位置插入一个元素—复杂度 $O(N)$

利用 `list.delete` 或 `del` 删除一个元素—复杂度 $O(N)$

操作	复杂度
复制	$O(N)$
添加元素(在尾部添加)	$O(1)$
插入元素(在指定位置插入)	$O(N)$
获取元素	$O(1)$
修改元素	$O(1)$
删除元素	$O(N)$
遍历	$O(N)$
获取长度为 k 的切片	$O(k)$
删除切片	$O(N)$
列表扩展	$O(k)$
测试是否在列表中	$O(N)$
<code>min()/max()</code>	$O(n)$
获取列表长度	$O(1)$

列表推导

要习惯用列表推导，因为这更加高效和简短，涉及的语法元素少。在大型的程序中，这意味着更少的错误，代码也更容易阅读。

列表推导

要习惯用列表推导，因为这更加高效和简短，涉及的语法元素少。在大型的程序中，这意味着更少的错误，代码也更容易阅读。

```
[i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

其它习语

1.使用 `enumerate`.在循环使用序列时，这个内置函数可以方便的获取其索引：

```
for i, element in enumerate(['one', 'two', 'three']):
    print(i, element)
result:
0 one
1 two
2 three
```

2.如果需要一个一个合并多个列表中的元素，可以使用 `zip()`。对两个大小相等的可迭代对象进行均匀遍历，这是一个非常常用的模式：

```
for item in zip([1, 2, 3], [4, 5, 6]):
    print(item)
(1, 4)
(2, 5)
(3, 6)
```

3.序列解包

#带星号的表达式可以获取序列的剩余部分

```
first, second, *reset = 0, 1, 2, 3
first
0
second
1
reset
[2, 3]
1
```

字典

字典是 python 中最通用的数据结构之一。dict 可以将一组唯一的键映射到相应的值。

我们也可以用前面列表推导的方式来创建一个字典。

```
squares = {number: number**2 for number in range(10)}
print(squares)
result:
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

在遍历字典元素时，有一点需要特别注意。字典里的 `keys()`，`values()` 和 `items()` 3 个方法的返回值不再是列表，而是视图对象（view objects）。

`keys()`: 返回 `dict_keys` 对象，可以查看字典所有键

`values()`: 返回 `dict_values` 对象，可以查看字典的所有值

`items()`: 返回 `dict_items` 对象，可以查看字典所有的 {key, value} 二元元组。

视图对象可以动态查看字典的内容，因此每次字典发生变化时，视图都会相应的改变，见下面这个例子：

```
words = {'foo': 'bar', 'fizz': 'bazz'}
items = words.items()
words['spam'] = 'eggs'
print(items)
result:
dict_items([('foo', 'bar'), ('fizz', 'bazz'), ('spam', 'eggs')])
```

视图无需冗余的将所有值都保存在内存中，像列表那样。但你仍然可以获取其长度（使用 `len`），也可以测试元素是否包含在其中（使用 `in` 子句）。当然，视图是迭代的。

实现细节

CPython 使用伪随机探测(pseudo-random probing)的散列表(hash table)作为字典的底层数据结构。由于这个实现细节，只有可哈希的对象才能作为字典的键。

Python 中所有不可变的内置类型都是可哈希的。可变类型（如列表，字典和集合）就是不可哈希的，因此不能作为字典的键。

字典的三个基本操作（添加元素，获取元素和删除元素）的平均事件复杂度为 $O(1)$ ，但是他们的平摊最坏情况复杂度要高得多，为 $O(N)$ 。

操作	平均复杂度	平摊最坏情况复杂度
获取元素	$O(1)$	$O(n)$
修改元素	$O(1)$	$O(n)$
删除元素	$O(1)$	$O(n)$

复制	$O(n)$	$O(n)$
遍历	$O(n)$	$O(n)$

还有一点很重要，在复制和遍历字典的操作中，最坏的复杂度中的 n 是字典曾经达到的最大元素数目，而不是当前的元素数目。换句话说，如果一个字典曾经元素个数很多，后来又大大减小了，那么遍历这个字典可能会花费相当长的事件。因此在某些情况下，如果需要频繁的遍历某个词典，那么最好创建一个新的字典对象，而不是仅在旧字典中删除元素。

字典的缺点和替代方案

使用字典的常见陷阱就是，它并不会按照键的添加顺序来保存元素的顺序。在某些情况下，字典的键是连续的，对应的散列值也是连续值（例如整数），那么由于字典的内部实现，元素的实现可能和添加的顺序相同：

```
keys = {num: None for num in range(5)}.keys()
print(keys)
result:
dict_keys([0, 1, 2, 3, 4])
但是，如果散列方法不同的其它数据类型，那么字典就不会保存元素顺序。
age = {str(i): i for i in range(100)}
keys = age.keys()
print(keys)
result:
dict_keys(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14',
', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29',
', '30', '31', '32', '33', '34', '35', '36', '37', '38', '39', '40', '41', '42', '43', '44',
', '45', '46', '47', '48', '49', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '
60', '61', '62', '63', '64', '65', '66', '67', '68', '69', '70', '71', '72', '73', '74', '7
5', '76', '77', '78', '79', '80', '81', '82', '83', '84', '85', '86', '87', '88', '89', '90
', '91', '92', '93', '94', '95', '96', '97', '98', '99'])
```

理论上，键的顺序不应该是这样的，应该是乱序。。。具体为什么这样，等以后明白了再补充

如果我们需要保存添加顺序怎么办？python 标准库的 `collections` 模块提供了名为 `OrderedDict` 的有序字典。

集合

集合是一种鲁棒性很好的数据结构，当元素顺序的重要性不如元素的唯一性和测试元素是否包含在集合中的效率时，大部分情况下这种数据结构极其有用。

python 的内置集合类型有两种：

`set()`：一种可变的、无序的、有限的集合，其元素是唯一的、不可变的（可哈希的）对象。
`frozenset()`：一种不可变的、可哈希的、无序的集合，其元素是唯一的，不可变的哈希对象。
`set([set([1, 2, 3]), set([2, 3, 4])])`

result:

Traceback (most recent call last):

```
File "/pycharm_project/LearnPython/Part1/demo.py", line 1, in <module>
    set([set([1, 2, 3]), set([2, 3, 4])])
TypeError: unhashable type: 'set'
```

```
set([frozenset([1, 2, 3]), frozenset([2, 3, 4])])
```

result:不会报错

set 里的元素必须是唯一的，不可变的。但是 set 是可变的，所以 set 作为 set 的元素会报错。

实现细节

CPython 中集合和字典非常相似。事实上，集合被实现为带有空值的字典，只有键才是实际的集合元素。此外，集合还利用这种没有值的映射做了其它的优化。

由于这一点，可以快速的向集合中添加元素、删除元素、检查元素是否存在。平均时间复杂度为 $O(1)$ ，最坏的事件复杂度是 $O(n)$ 。

6. 垃圾回收相关问题。

python 采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

python 中有自动内存回收机制，一般情况不需要程序员来处理。

gc 方式 1：引用计数

若此对象无其他对象引用，则立马回收掉

优点：简单、实时（将处理垃圾时间分摊到运行代码时，而不是等到一次回收）

缺点：

- 1.保存对象引用数会占用一点点内存空间
- 2.每次执行语句都可能更新引用数，不再使用大的数据结构时，会引起大量对象被回收
- 3.不能处理循环引用的情况

gc 方式 2：标记-清除（Mark—Sweep）

此方式主要用来处理循环引用的情况，只有容器对象（list、dict、tuple，instance）才会出现循环引用的情况

循环引用示例：

处理过程

- 1.将所有容器对象放到一个双向链表中（链表为了方便插入删除），这些对象为 0 代
- 2.循环遍历链表，如果被本链表内的对象引入，自身的被引用数-1，如果被引用数为 0，则触发引用计数回收条件，被回收掉
- 3.未被回收的对象，升级为 1 代

触发条件：

因为循环引用的原因，并且因为你的程序使用了一些比其他对象存在时间更长的对象，从而被分配对象的计数值与被释放对象的计数值之间的差异在逐渐增长。一旦这个差异累计超过某个阈值，则 Python 的收集机制就启动了，并且触发上边所说的零代算法，释放“浮动的垃圾”，并且将剩下的对象移动到一代链表。

随着时间的推移，一代链表越来越多，多到触发 gc 阈值，同样会对一代链表进行标记清除操作，然后将剩下活跃对象升为二代。

何时触发

- 1.被引用为 0 时，立即回收当前对象

2.达到了垃圾回收的阈值，触发标记-清除

3.手动调用 `gc.collect()`

4.Python 虚拟机退出的时候

内存池

Python 为了避免频繁的申请和删除内存所造成系统切换于用户态和核心态的开销，从而引入了**内存池机制**，专门用来管理**小内存**的申请和释放。整个小块内存的内存池可以视为一个层次结构，其一共分为 4 层，从下之上分别是 **block**、**pool**、**arena** 和内存池。需要说明的是：**block**、**pool** 和 **area** 都是代码中可以找到的实体，而最顶层的内存池只是一个概念上的东西，表示 Python 对于整个小块内存分配和释放行为的内存管理机制。

注意，内存大小以 256 字节为界限，大于则通过 `malloc` 进行分配，小于则通过内存池分配。

垃圾回收

Python 的 GC 模块主要运用了引用计数来跟踪和回收垃圾。在引用计数的基础上，还可以通过“标记—清除”解决容器对象可能产生的**循环引用**的问题。通过分代回收以空间换取时间进一步提高垃圾回收的效率。

引用计数

原理：当一个对象的引用被创建或者复制时，对象的引用计数加 1；当一个对象的引用被销毁时，对象的引用计数减 1，当对象的引用计数减少为 0 时，就意味着对象已经再没有被使用了，可以将其内存释放掉。

优点：引用计数有一个很大的优点，即实时性，任何内存，一旦没有指向它的引用，就会被立即回收，而其他的垃圾收集技术必须在某种特殊条件下才能进行无效内存的回收。

缺点：但是它也有弱点，引用计数机制所带来的维护引用计数的额外操作与 Python 运行中所进行的内存分配和释放，引用赋值的次数是成正比的，这显然比其它那些垃圾收集技术所带来的额外操作只是与待回收的内存数量有关的效率要低。同时，引用技术还存在另外一个很大的问题—循环引用，因为对象之间相互引用，每个对象的引用都不会为 0，所以这些对象所占用的内存始终都不会被释放掉。

标记—清除

标记—清除只关注那些可能会产生循环引用的对象，显然，像是 `PyIntObject`、`PyStringObject` 这些不可变对象是不可能产生循环引用的，因为它们内部不可能持有其它对象的引用。Python 中的循环引用总是发生在 **container** 对象之间，也就是能够在内部持有其它对象的对象，比如 `list`、`dict`、`class` 等等。这也使得该方法带来的开销只依赖于 **container** 对象的数量的数量？？？

原理：1. 寻找跟对象（**root object**）的集合作为垃圾检测动作的起点，跟对象也就是一些全局引用和函数栈中的引用，这些引用所指向的对象是不可被删除的；2. 从 **root object** 集合出发，沿着 **root object** 集合中的每一个引用，如果能够到达某个对象，则说明这个对象是可达的，那么就不会被删除，这个过程就是垃圾检测阶段；3. 当检测阶段结束以后，所有的对象就分成可达和不可达两部分，所有的可达对象都进行保留，其它的不可达对象所占用的内存将会被回收，这就是垃圾回收阶段。（底层采用的是**链表**将这些集合的对象连接在一起）

缺点：标记和清除的过程效率不高。

分代回收

原理：将系统中的所有内存块根据其存活时间划分为不同的集合，每一个集合就成为一个“代”，Python 默认定义了三代对象集合，垃圾收集的频率随着“代”的存活时间的增大而减小。也就是说，活得越长的对象，就越不可能是垃圾，就应该减少对它的垃圾收集频率。那么如何来衡量这个存活时间：通常是利用几次垃圾收集动作来衡量，如果一个对象经过的垃圾收集次数越多，可以得出：该对象存活时间就越长。

7. 并发编程的相关问题。

并发：在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行，但任一个时刻点上只有一个程序在处理机上运行。简言之，是指系统具有处理多个任务的能力。

并行：当系统有一个以上 CPU 时，则线程的操作有可能非并发。当一个 CPU 执行一个线程时，另一个 CPU 可以执行另一个线程，两个线程互不抢占 CPU 资源，可以同时进行，这种方式我们称之为并行 (Parallel)。简言之，是指系统具有同时处理多个任务的能力。

对于一次 IO 访问（以 read 举例），数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。所以说，当一个 read 操作发生时，它会经历两个阶段：

1. 等待数据准备 (Waiting for the data to be ready)
2. 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

同步：当进程执行 IO(等待外部数据)的时候，-----等。同步（例如打电话的时候必须等）

异步：当进程执行 IO(等待外部数据)的时候，-----不等，去执行其他任务，一直等到数据接收成功，再回来处理。异步（例如发短信）

当我们去爬取一个网页的时候，要爬取多个网站，有些人可能会发起多个请求，然后通过函数顺序调用。执行顺序也是先调用先执行。效率非常低。

调用 blocking IO 会一直 block 住对应的进程直到操作完成，而 non-blocking IO 在 kernel 还准备数据的情况下会立刻返回。

只要有一丁点阻塞，就是阻塞 IO。

异步 IO 的特点就是全程无阻塞。

有些人常把同步阻塞和异步非阻塞联系起来，但实际上经过分析，阻塞与同步，非阻塞和异步的定义是不一样的。同步和异步的区别是遇到 IO 请求是否等待。阻塞和非阻塞的区别是数据没准备好的情况下是否立即返回。同步可能是阻塞的，也可能是非阻塞的，而非阻塞的有可能是同步的，也有可能是异步的。阻塞和非阻塞，描述的是一种状态，而同步与非同步描述的是行为方式。

GIL(全局解释器锁)：无论你启多少个线程，你有多少个 cpu，Python 在执行的时候会淡定的在同一时刻只允许一个线程运行。

8. 协程和异步 I/O 相关知识。

- 协程

协程，Coroutine，是一种用户态的轻量级线程。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方（非 CPU），在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，CPU 感觉不到协程的存在，协程是用户自己控制的。

协程实现原理：利用一个线程，分解一个线程成为多个“微线程”，属于程序级别的划分。

协程的优点：

无需线程上下文切换的开销
无需数据操作锁定及同步的开销
方便切换控制流，简化编程模型
高并发+高扩展性+低成本：一个 CPU 支持上万的协程都不是问题。所以很适合用于高并发处理

缺点：

无法利用多核资源：协程的本质是个单线程,它不能同时将单个 CPU 的多个核用上,协程需要和进程配合才能运行在多 CPU 上，协程如果要使用多核 CPU 的话，那么就需要先启多个进程，在每个进程下启一个线程，然后在线程下在启协程。

日常所编写的绝大部分应用都没有这个必要，除非是 cpu 密集型应用。

python 通过 `yield` 提供对协程的基本支持，但是不完全，第三方库 `gevent` 为协程提供了比较完善的支持。

从句法上看，协程与生成器类似，都是定义体中包含 `yield` 关键字的函数。可是，在协程中，`yield` 通常出现在表达式的右边（例如，`datum = yield`），可以产出值，也可以不产出 — 如果 `yield` 关键字后面没有表达式，那么生成器产出 `None`。

参考网址：<https://blog.csdn.net/andybegin/article/details/77884645>

协程可能会从调用方接收数据，不过调用方把数据提供给协程使用的是 `.send(datum)` 方法，而不是 `next(...)` 函数。

```
def simple_coroutine():
    print('-> start')
    x = yield
    print('-> recived', x)

sc = simple_coroutine()

next(sc)
sc.send('zhexiao')
```

1. 协程使用生成器函数定义：定义体中有 `yield` 关键字。
2. `yield` 在表达式中使用；如果协程只需从客户那里接收数据，那么产出的值是 `None` —— 这个值是隐式指定的，因为 `yield` 关键字右边没有表达式。
3. 首先要调用 `next(...)` 函数，因为生成器还没启动，没在 `yield` 语句处暂停，所以一开始无法发送数据。
4. 调用 `send` 方法，把值传给 `yield` 的变量，然后协程恢复，继续执行下面的代码，直到运行到下一个 `yield` 表达式，或者终止。

==注意：`send` 方法只有当协程处于 `GEN_SUSPENDED` 状态下时才会运作，所以我们使用 `next()` 方法激活协程到 `yield` 表达式处停止，或者我们也可以使用 `sc.send(None)`，效果与 `next(sc)` 一样==。

协程可以身处四个状态中的一个。当前状态可以使用 `inspect.getgeneratorstate(...)` 函数确定，该函数会返回下述字符串中的一个：

1. `GEN_CREATED`：等待开始执行
2. `GEN_RUNNING`：解释器正在执行

```

3. GEN_SUSPENDED: 在 yield 表达式处暂停
4. GEN_CLOSED: 执行结束
import time
def consumer(name):
    print('--->staring eating baozi....')
    while True:
        new_baozi = yield
        print("%s is eating baozi %s" %(name,new_baozi))
def producer():
    r = con.__next__()
    r = con2.__next__()
    n = 0
    while n<5:
        n += 1
        print("chenkaifang is making baozi %s" %n)
        con.send(n)
        con2.send(n)
if __name__ == '__main__':
    con = consumer("c1")
    con2 = consumer("c2")
    p = producer()
yield from 高级用法....
greenlet 有已经封装好的协程:
from greenlet import greenlet
def test1():
    print(12)
    gr2.switch()
    print(34)
    gr2.switch()
def test2():
    print(56)
    gr1.switch()
    print(78)
gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()

```

参考网址: <https://www.cnblogs.com/zhaof/p/7536569.html>

Gevent 是一种基于协程的 Python 网络库，它用到 **Greenlet** 提供的，封装了 **libevent** 事件循环的高层同步 API。它让开发者在不改变编程习惯的同时，用同步的方式写异步 I/O 的代码。它的特点是协程的自动切换（遇到 IO 操作自动轮训切换）。

使用 **Gevent** 的性能确实要比用传统的线程高，甚至高很多。但这里不得不说它的一个坑，实际使用中要慎用 **Gevent**：

Monkey-patching，我们都叫猴子补丁，因为如果使用了这个补丁，**Gevent** 直接修改标准库里面大部分的阻塞式系统调用，包括 **socket**、**ssl**、**threading** 和 **select** 等模块，而变为协作式运行。但是我们无法保证你在复杂的生产环境中有哪些地方使用这些标准库会由于打了补丁而出现奇怪的问题

第三方库支持。得确保项目中用到其他用到的网络库也必须使用纯 Python 或者明确说明支持 **Gevent**
Greenlet 不支持 **Jython** 和 **IronPython**，这样就无法把 **gevent** 设计成一个标准库了

`gevent` 是第三方库，通过 `greenlet` 实现协程，其基本思想是：当一个 `greenlet` 遇到 IO 操作时，比如访问网络，就自动切换到其他的 `greenlet`，等到 IO 操作完成，再在适当的时候切换回来继续执行。由于 IO 操作非常耗时，经常使程序处于等待状态，有了 `gevent` 为我们自动切换协程，就保证总有 `greenlet` 在运行，而不是等待 IO。由于切换是在 IO 操作时自动完成，所以 `gevent` 需要修改 Python 自带的一些标准库，这一过程在启动时通过 `monkey patch` 完成。

`gevent` 使用实例 1:

```
from gevent import monkey;monkey.patch_socket()
import gevent
def f(n):
    for i in range(n):
        print(gevent.getcurrent(),i)
        gevent.sleep(0)#让出 CPU 使用权
g1 = gevent.spawn(f,5)
g2 = gevent.spawn(f,5)
g3 = gevent.spawn(f,5)
g1.join()
g2.join()
g3.join()
gevent I/O 操作实例 1:
from gevent import monkey;monkey.patch_all()
import gevent
import urllib.request

def f(url):
    print("GET:%s" %url)
    resp = urllib.request.urlopen(url)
    data = resp.read()
    print("%d bytes received form %s" %(len(data),url))
gevent.joinall([
    gevent.spawn(f,"https://www.python.org/"),
    gevent.spawn(f,"https://www.yahoo.com/"),
    gevent.spawn(f,"https://github.com/"),
])
```

- 多线程

由于 GIL 的存在，即使硬件有 N 个核，也只能利用一个核。

`ThreadLocal` 类型处理变量共享与绑定，与 Java 类似。

`threading.Thread` 类的使用:

- 1, 在自己的线程类的 `__init__` 里调用 `threading.Thread.__init__(self, name = threadname)` `Threadname` 为线程的名字
- 2, `run()`，通常需要重写，编写代码实现需要的功能。
- 3, `getName()`，获得线程对象名称
- 4, `setName()`，设置线程对象名称
- 5, `start()`，启动线程
- 6, `join([timeout])`，等待另一线程结束后再运行。
- 7, `setDaemon(bool)`，设置子线程是否随主线程一起结束，必须在 `start()` 之前调用。默认为 `False`。
- 8, `isDaemon()`，判断线程是否随主线程一起结束。
- 9, `isAlive()`，检查线程是否在运行中。

参考网址:

<https://www.cnblogs.com/chengd/articles/7770898.html>

<https://www.cnblogs.com/semiok/articles/2640929.html>

#Lock 不允许同一线程多次 acquire

```
import time
import threading
balance = 0
lock = threading.Lock()
def change(n):
    global balance
    balance = balance + n
    balance = balance - n
def run_thread(n):
    for i in range(10000):
        lock.acquire()
        try:
            change(n)
        finally:
            lock.release()
t1 = threading.Thread(target = run_thread,args=(5,))
t2 = threading.Thread(target = run_thread,args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

Rlock 允许同一线程多次 acquire

threading.Condition, 高级锁, 内部维护一个锁对象, 提供 wait、notify、notifAll 方法 (必须在获得锁后才能调用)

threading.Semaphore BoundedSemaphore 信号量同步

Rlock 允许同一线程多次 acquire

threading.Condition, 高级锁, 内部维护一个锁对象, 提供 wait、notify、notifAll 方法 (必须在获得锁后才能调用)

threading.Semaphore BoundedSemaphore 信号量同步

thread.Event: 事件处理机制

Queue: 同步队列

threading.active_count threading.current_thread threading.enumerate threading.get_id
ent threading.main_thread

线程同步的 4 种方式: 锁、信号量、条件变量、同步队列

- 多进程

Windows 下 python 用 multiprocessing 实现多进程, multiprocessing 模块不但支持多进程, 其中 managers 子模块还支持把多进程分布到多台机器上, 一个服务进程可以作为调度者, 将任务分布到其他多个进程中, 依靠网络通信。由于 managers 模块封装很好, 不必了解网络通信的细节, 就可以很容易地编写分布式多进程程序。

参考网址: <https://blog.csdn.net/cityzenoldwang/article/details/78584175>

```
import multiprocessing
import os
def run_proc(name):
    print('Child process {0} {1} Running '.format(name, os.getpid()))
if __name__ == '__main__':
    print('Parent process {0} is Running'.format(os.getpid()))
    for i in range(5):
        p = multiprocessing.Process(target=run_proc, args=(str(i),))
        print('process start')
        p.start()
    p.join()
    print('Process close')
```

Pool 可以提供指定数量的进程供用户使用, 默认是 CPU 核数。当有新的请求提交到 Pool 的时候, 如果池子没有满, 会创建一个进程来执行, 否则就会让该请求等待。

apply_async 方法用来同步执行进程, 允许多个进程同时进入池子, apply 只能允许一个进程进入池子, 在一个进程结束之后, 另外一个进程才可以进入池子。

进程间通信 Queue、Pipe、Socket。基于消息传递的并发编程是大势所趋, 通过消息队列交换数据。这样极大地减少了对使用锁定和其他同步手段的需求, 还可以扩展到分布式系统中。

multiprocessing 给每个进程赋予单独的 Python 解释器, 这样就规避了全局解释锁所带来的问题。

• 异步 IO

通常, 我们写服务器处理模型的程序时, 有以下几种模型:

- (1) 每收到一个请求, 创建一个新的进程, 来处理该请求;
- (2) 每收到一个请求, 创建一个新的线程, 来处理该请求;
- (3) 每收到一个请求, 放入一个事件列表, 让主进程通过非阻塞 I/O 方式来处理请求 (就是以事件驱动的方式来处理)

上面的几种方式, 各有千秋,

第 (1) 种方法, 由于创建新的进程的开销比较大, 所以, 会导致服务器性能比较差, 但实现比较简单。

第 (2) 种方式, 由于要涉及到线程的同步, 有可能会面临死锁等问题。

第 (3) 种方式, 在写应用程序代码时, 逻辑比前面两种都复杂。

综合考虑各方面因素, 一般普遍认为第 (3) 种方式是大多数网络服务器采用的方式。

对于一次 IO 访问 (以 read 举例), 数据会先被拷贝到操作系统内核的缓冲区中, 然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。所以说, 当一个 read 操作发生时, 它会经历两个阶段:

等待数据准备 (Waiting for the data to be ready) (就是将数据放到内核缓存中)

将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

基于这两步, Linux 有 5 种网络模型: 阻塞、非阻塞、I/O 多路复用、信号驱动、异步 IO。

IO multiplexing 就是我们说的 select, poll, epoll。I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符, 而这些文件描述符 (socket 连接) 其中的任意一个进入就绪状态, select () 函数就可以返回。epoll 同样只告知那些就绪的文件描述符, epoll 采用基于事件的就绪通知方式, epoll 会告诉用户进程具体哪个 socket 连接有数据了, 所以用户进程不需要在将所有 socket 连接全都循环一次才发现具体哪个有数据。epoll 解决 C10K 问题, 采用 mmap 减少复制开销, epoll 技术的编程模型就是异步

非阻塞回调，也可以叫做 **Reactor**、事件驱动、事件轮循（**EventLoop**）、**libevent**、**Tornado**、**Node.js** 这些就是 **epoll** 时代的产物。

协程本质上也是异步非阻塞技术，它是将事件回调进行了包装，让程序员看不到里面的事件循环。进程/线程是操作系统充当了 **EventLoop** 调度，而协程是自己用 **epoll** 进行调度。

asyncio 在 **python3.4** 被引入标准库，**Python 3.5** 添加了 **async** 和 **await** 这两个关键字，分别用来替换 **asyncio.coroutine** 和 **yield from**，协程成为新的语法，而不再是一种生成器类型了。

首先需要明确一点，**asyncio** 使用单线程、单个进程的方式切换；现存的一些库其实并不能原生的支持 **asyncio**（因为会发生阻塞或者功能不可用），比如 **requests**，如果要写爬虫，配合 **asyncio** 的应该用 **aiohttp**，其他的如数据库驱动等各种 **Python** 对应的库也都得使用对应的 **aioXXX** 版本了。

asyncio 模块包含多种同步机制：信号量、锁、条件变量、事件、队列

```
import asyncio

async def hello1():
    print("1,Hello World!")
    r = await asyncio.sleep(1)
    print("1,Hello again!")
    for i in range(5):
        print(i)

async def hello2():
    print("2,Hello World!")
    print("2,Hello again!")
    for i in range(5):
        print(i)

coros = [hello1(),hello2()]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(*coros))
```

- 同步、异步、阻塞、非阻塞

同步：就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。

异步：当一个异步过程调用发出后，调用者不会立刻得到结果。通过状态、通知来通知调用者，或通过回调函数处理这个调用。

单进程的异步编程模型称为协程。

阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在得到结果之后才会返回。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。

非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

同步阻塞：效率最低，什么都不能干；异步阻塞；同步非阻塞也效率低；异步非阻塞效率高。

- 进程、线程、协程应用场景

IO 密集型：用多线程+**gevent**（更好），多线程（抛弃 **Gevent**，用 **asyncio**）

计算密集型：用多进程

如果要启动大量的子进程，可以用进程池的方式批量创建子进程。`ThreadPoolExecutor` 和 `ProcessPoolExecutor`

如果分不清任务是 CPU 密集型还是 IO 密集型，就用下面两个方法试，哪个快用哪个：

```
from multiprocessing import Pool
from multiprocessing.dummy import Pool
```

如果一个任务拿不准是 CPU 密集还是 I/O 密集型，且没有其它不能选择多进程方式的因素，都统一直接上多进程模式。

aiohttp。一个实现了 PEP3156 的 HTTP 的服务器，且包含客户端相关功能。最早出现，应该最知名。

sanic。后起之秀，基于 Flask 语法的异步 Web 框架。

uvloop。用 Cython 编写的、用来替代 `asyncio` 事件循环。作者说「它在速度上至少比 `Node.js`、`gevent` 以及其它任何 Python 异步框架快 2 倍」。

ujson。比标准库 `json` 及其社区版的 `simplejson` 都要快的 JSON 编解码库。

Python 三器

可迭代对象、生成器、迭代器三者的关系

1. 迭代器一定是可迭代对象
2. 生成器是迭代器的一种
3. 可迭代对象：必须实现 `__iter__` 方法
4. 迭代器：必须实现 `__iter__` 方法 和 `__next__` 方法
5. 生成器：必须实现 `__iter__` 方法 和 `__next__` 方法，`yield` 代替了这两个方法
6. 工具包： `from collections import Iterable, Iterator`
7. 查看抽象接口：

```
In [265]: Iterable.__abstractmethods__
Out[265]: frozenset({'__iter__'})
In [266]: Iterator.__abstractmethods__
Out[266]: frozenset({'__next__'})
```

8. 可迭代对象都可以被 `for` 循环 所遍历， 另外 实现了 `__getitem__` 的类 其对象也可 `for` 遍历
关于 `__getitem__` 等魔法方法，以后会单独写一篇文章。

Python 之路(四) — Python 三器(迭代器、生成器、装饰器)

一、迭代器(Iterator)

可迭代对象 (Iterable Object)

表面来看，只要可以用 `for...in...` 进行遍历元素的对象就是可迭代对象

语法层面，如果一个对象实现了 `__iter__` 方法，那么这个对象就是可迭代对象

可迭代对象 vs 迭代器

可迭代对象：实现了 `__iter__` 方法；迭代器：实现了 `__next__` 方法

可以被 `next()` 函数调用并不断返回下一个值的对象称为迭代器 (Iterator)，直到没有数据时抛出

`StopIteration` 错误。不断 `next` 取下一个值的数据流对象。可以看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过 `next()` 函数实现按需计算下一个数据，所以 `Iterator` 的计算是惰性的，只有在需要返回下一个数据时它才会计算。

迭代器一定是可迭代对象，但可迭代对象不一定是迭代器。如： `[1,3,4]` 可以迭代，但不是迭代器

二、生成器(Generator)

定义：特殊的迭代器，仅保存计算规则不提前生成结果，需要(`next` 方法调用)时进行计算生成

生成器

生成器有两种写法：

形式一：

```
In [230]: def f():
...:     for x in range(10):
```

```
...:         yield x
```

```
In [231]: y = f()
```

形式二:

```
In [239]: y = ( x for x in range(10) )
```

```
In [240]: y
```

```
Out[240]: <generator object <genexpr> at 0x0000024A4D3AFB48>
```

'推动生成器' 迭代有三种方式:

方式 1:

```
In [232]: next(y)
```

```
Out[232]: 0
```

方式 2:

```
In [233]: y.__next__()
```

```
Out[233]: 1
```

方式 3:

```
In [236]: y.send(None)
```

```
Out[236]: 2
```

前两种方式是等价的, 第三种方式有些差别, 且听我说:

```
In [248]: def f():
```

```
...:     print(1)
```

```
...:     a = yield 2
```

```
...:     print(a)
```

```
In [249]: y = f()
```

```
In [250]: y.send(None)
```

```
1      # 这里是 print(1)的结果
```

```
Out[250]: 2
```

```
In [251]: y.send(100)
```

```
100    # 这里是 print(a)的结果
```

不知道阁下能否看出和两种的差别:

1. `yield 2` # 这行多了个赋值操作

2. `send(100)` # `send()` 函数里面放了个, 然后 `print(a)` 打印的就是 100

举个例子:

1. `a = yield 2` # 相当于一个士兵等待指令, 等号左边还未执行, 程序就被封锁了

2. `send(100)` # 长官输入了一个 100, 士兵收到后就把程序解封, 并执行等号左边

3. 于是 就相当于 `a` 被赋值为 100

2. 生成器

生成器: 按照一定规则, 生成数据的对象

(1) 普通函数

获取 1~10 的数字的函数

```
def g_number():
```

```
    ls = list()
```

```
    for i in range(10):
```

```
        ls.append(i)
```

```
    return ls
```

优点: 简单, 明了! 缺点: 一次性获取所有的数据, 加载内存, 浪费空间。

(2) 生成器函数

```
def g_number():
```

```
    for i in range(10):
```

```
        yield i
```

`g = g_number()` # 获取的是一个生成器对象 `<generator object>`

`print(next(g))` # 获取生成器产生的数据

`print(g.__next__())` # 获取生成器产生的数据

优点：生成器对象每次执行 `next()` 都会获取一次数据，可以在项目开发过程中，按需获取

缺点：代码的可读性发生了提升

yield 关键字：让一个函数转换成了生成器对象，执行到 `yield` 会临时中断，等待 `next()` 调用后继续向下执行应用：

1. 生成一个构建计数器对象

2. 实现斐波那契数列

3. 列表生成器

4. `yield from` 从一个生成器中获取所有数据

用途：批量数据运算中（距离统计、大数据求平均、批量创建对象...）

迭代器

先看两个简单的函数：

```
In [294]: iter([1,2,3])
Out[294]: <list_iterator at 0x24a4e7f5780>
In [295]: reversed([1,2,3])
Out[295]: <list_reverseiterator at 0x24a4d38a6a0>
```

自定义迭代器：

```
In [283]: class A(Iterator): # 注意这里：继承了 Iterator 就不用实现 __iter__ 接口了
...:     def __init__(self, value):
...:         self.value = value
...:         self.index = -1
...:     def __next__(self): # 注意这里：__next__ 是写逻辑的主要接口，每次返回单个值
...:         self.index += 1
...:         return self.value[self.index]
```

```
In [284]: a = A(['Tom', 'Jerry'])
In [285]: next(a)
Out[285]: 'Tom'
In [286]: next(a)
Out[286]: 'Jerry'
```

串联合并迭代器：

1. 普通序列迭代器串联

```
In [287]: from itertools import chain
In [289]: chain(range(1),range(2),range(3))
Out[289]: <itertools.chain at 0x24a4d2c77f0>
In [290]: list(chain(range(1),range(2),range(3)))
Out[290]: [0, 0, 1, 0, 1, 2]
```

2. 字典序列迭代器串联

```
In [288]: from collections import ChainMap
In [291]: ChainMap({1:1},{2:2,3:3})
Out[291]: ChainMap({1: 1}, {2: 2, 3: 3})
In [292]: dict(ChainMap({1:1},{2:2,3:3}))
Out[292]: {1: 1, 2: 2, 3: 3}
```

迭代器的应用：

1. 自定义可迭代对象：计数器开发

2. 自定义斐波那契数列：迭代器对象实现

装饰器

装饰器如果按钻牛角尖的方式来理解的确是很头疼的事情。先说个例子吧：

```
In [296]: def f(func):
```

```

...:         def f1():
...:             print('原函数之前加点功能')
...:             func()          # 这就是原函数
...:             print('原函数之后加点功能')
...:     return f1

```

In [299]: @f #这句等价于=> func = f(func)

```

...:     def func():
...:         print('我是原函数哦')

```

In [300]: func()

```

>>     原函数之前加点功能
>>     我是原函数哦
>>     原函数之后加点功能

```

解释

用大白话来讲，装饰器就是在原函数的前后加功能。也就是给原函数加个外壳。看上去是调用的原函数，实则调用的是外壳函数，外壳函数里面包括原函数和一些其他的自定义功能

例子：

面包不好吃啊，但是你还想吃这个面包，咋整？上下抹点奶油，就变成了三明治（我没吃过。。）
吃其他面包的时候继续抹上奶油就行了（封装性）

面包-原函数

三明治-被装饰器装饰后的函数

各种功能（吃法）快速拼接：

@奶油

```

def 面包 1():
    pass

```

@奶油

```

def 面包 2():
    pass

```

@沙拉

```

def 面包 1():
    pass

```

@沙拉

```

def 面包 2():
    pass

```

存在问题：

@f #这句等价于=> func = f(func)

这是我上面说过的一句话，你仔细看看：

func 之前指向的是（原函数）面包的空间，恩，func 函数名 也就是(__name__) 是 func（面包）
现在他指向的是（新函数）三明治的空间，恩，它的函数名是 f1(三明治)

函数名变了，有些张冠李戴的感觉，如果不想让它变，并保持本身的函数名，看我操作：

In [301]: from functools import wraps

In [316]: def f(func):

```

...:     @wraps(func)    ### 没错 这里是最主要的，基本格式固定写法，照着写即可
...:     def f1():
...:         print(func.__name__)
...:         print('原函数之前加点功能')
...:         func()
...:         print('原函数之后加点功能')
...:     return f1

```

标准装饰器使用（原函数带有返回值 和 参数）：

```
In [323]: def f(func):
...:     @wraps(func)
...:     def f1(*args, **kwargs):
...:         print('报年龄和性别~~~')
...:         return func(*args, **kwargs)
...:     return f1
```

```
In [324]: @f                                     #每次都要记住 test=f(test) 这个隐含的变形，熟了就好了
...: def test(name,age):
...:     print(name, age)
```

```
In [325]: test('Tom',18)
```

-----下面是打印部分

报年龄和性别~~~

Tom 18

带参数的装饰器：

构造如下：

```
def foo(arg): # 其实就是在原来的基础上再再再次包装个外壳，仅此而已
    # 之前讲的装饰器 f 实现代码原封不动全放在这里面，看下面帮你写好了。
    def f(func):
        @wraps(func)
        def f1(*args, **kwargs):
            return func(*args, **kwargs)
        return f1
    return f
```

调用形如：

```
@foo('123') #多了个参数，可分解为 foo 的返回值 f 放在 @ 的后面，是不是一切又回到了从前？
def func():
    pass
```

装饰类：

应用场景：

Python WEB 框架 Flask/Django 都有 FBV 和 CBV 模式（以后我也会写这方面的文章）

FBV: Function Based View 简单来说，逻辑视图用函数来写

----那么需要装饰器的时候，直接用普通函数装饰器装饰到函数上即可

CBV: Class Based View 简单来说，逻辑视图用类来写

----这种没有函数，是用类写的，那么这时候就需要对类进行装饰了

核心理想：

还是记住上面讲的隐式变形，只不过这次传给装饰器的是类，对类的操作可就太多了。

里面一大堆黑魔法，下面我就来写一下类初始化功能性装饰器。（以后也会单独写黑魔法的文章）

```
In [1]: def f(c):
...:     def f1(*args, **kwargs):
...:         return c(*args, **kwargs)
...:     return f1
```

```
In [2]: @f
...: class A:
...:     def __init__(self,name):
...:         print(name)
```

```
In [3]: A('Tom')
```

```
Tom    # 这是 print 打印的输出
Out[3]: <__main__.A at 0x2948e8fb828>
```

类装饰器:

这个场景真没遇到过, 不过实现也很容易的。

```
In [8]: class A:
...:     def __init__(self,func):
...:         self.func = func
...:     def __call__(self,*args,**kwargs):
...:         print('类装饰填充了啊')
...:         self.func(*args, **kwargs)
...:
```

```
In [9]: @A
...: def test():
...:     print('我是原函数')
...:
```

```
In [10]: test()
类装饰填充了啊
我是原函数
```

说明: 后面关于类的装饰器如果理解困难当做了解即可, 用的也少。

3. 装饰器

闭包: 函数中嵌套声明的函数, 就是闭包函数

装饰器:

基本语法:

```
def 装饰器名称 (fn):
    def wrapper(*args,**kwargs):
        目标函数执行前, 要添加的功能代码
        res = fn(*args,**kwargs)
        目标函数执行后, 要添加的功能代码
        return res
    return wrapper
```

调用方式:

@装饰器

```
def register():
    """用户注册的函数"""
```

```
register()
```

应用: 身份认证、权限授权、资源访问、日志记录、时间统计等等。

python 三大器

- 迭代器: 有 `iter()` 和 `next()` 方法
- 生成器: 函数中将 `return` 换成 `yield`
- 装饰器: 闭包的本质

迭代器

```
...
```

遇到问题没人解答? 小编创建了一个 Python 学习交流 QQ 群: 579817333
寻找志同道合的小伙伴, 互帮互助, 群里还有不错的视频学习教程和 PDF 电子书!

```
...
```

```

# 迭代器 拥有__iter__()和__next__()方法
# 先将可迭代对象赋值一个新的变量转成迭代器
# 每次都返回一个值
# 一个__next__()对应一个输出结果
s = "1234"
new_s = s.__iter__()
print(new_s.__next__())# 输出 1
print(new_s.__next__())# 输出 2

# 变量名 = 可迭代对象
# 新赋值的变量名 = 变量名.__iter__() # 装成迭代器
# print(新赋值的变量名.__next__())

```

生成器

```

# 生成器 函数中将 return 改成 yield 的函数
def func():
    print(1)
    yield 2
g = func()
print(next(g))

# def 函数名():
#     print(输出)
#     yield 返回值
# 新的函数名 = 调用函数名()
# print(next(新的函数名))

# yield from 将可迭代对象逐行输出 节省空间

```

装饰器

```

...

```

遇到问题没人解答？小编创建了一个 Python 学习交流 QQ 群：579817333
寻找志同道合的小伙伴，互帮互助，群里还有不错的视频学习教程和 PDF 电子书！

```

...

```

```

# 闭包
def func():
    a = 11
    def foo():
        print(a)
    return foo
ret = func()
print(ret.__closure__)# 返回内存地址

```

```

def func():
    a = 11
    def foo():
        print(a)
    return foo
func()()
print(func.__closure__)# 返回 None
# 递归 不断自己调用自己，有明确终止条件
def age(n):

```

```

    if n == 4:
        return 18
    else:
        return age(n+1)-2
print(age(1))# 输出 12
# 将 age"1"的实参传给 age"n"形参
# if 1!=4,执行 else
# else return age(1+1)=2
# 直到 4==4, return18
# 18 反给 age(3),18-2=16
# 一直反 age(2),16-2=14 age(1),14-12=12
# age(1)=12
# 装饰器 在不改变源代码和调用方法的基础上增加新功能
# 将闭包的参数换成函数名
def func(f):
    def foo(*args,**kwargs):
        print("装饰器")
        return f()
    return foo
@func
# @函数名 语法糖 相当于 f=func(f)
def index():
    print(123)
    return 456
print(index())
# 输出 装饰器 123 456

# def 外层函数(参数):
#     def 内层函数(动态参数,形参):
#         print()
#         return 参数()
#     return 内层函数
# @外层函数
# def 源函数():
#     print()
#     return
# print(源函数())

```

Django 和 Flask

01、MVC 架构（MTV）解决了什么问题。

02、中间件的执行流程以及如何自定义中间件。

03、REST 数据接口如何设计（URL、域名、版本、过滤、状态码、安全性）。

建议阅读阮一峰的《[RESTful API 设计指南](#)》。

04、使用 ORM 框架实现 CRUD 操作的相关问题。

- 如何实现多条件组合查询 / 如何执行原生的 SQL / 如何避免 N+1 查询问题

05、如何执行异步任务和定时任务。

06、如何实现页面缓存和查询缓存？缓存如何预热？

Django 和 Flask:

1. MVC 架构 (MTV) 解决了什么问题。

MVC 与 MTV 的理解与区别

MVC (Model View Controller 模型-视图-控制器)

是一种 Web 架构的模式。特点: 把业务逻辑、模型数据、用户界面分离开来, 让开发者将数据与表现解耦。

Model: 代表数据存取层,

View 代表的是系统中选择显示什么和怎么显示的部分,

Controller 指的是系统中根据用户输入并视需要访问模型, 以决定使用哪个视图的那部分。

MTV (Model Templates View 模型-模板-视图):

1. Models: 数据存取层。该层处理与数据相关的所有事务: 如何存取、如何验证有效, 是一个抽象层, 用来构建和操作你的 web 应用中的数据, 模型是你的数据的唯一的、权威的信息源。它包含你所储存数据的必要字段和行为。通常, 每个模型对应数据库中唯一的一张表。

2. 模板(templates): 即表现层。该层处理与表现相关的决定: 如何在页面或其他类型文档中进行显示。模板层提供了设计友好的语法来展示信息给用户。使用模板方法可以动态地生成 HTML。模板包含所需 HTML 输出的静态部分, 以及一些特殊的语法, 描述如何将动态内容插入。

3. 视图 (views): 业务逻辑层, 该层包含存取模型及调取恰当模板的相关逻辑。用于封装负责处理用户请求及返回响应的逻辑。视图可以看作是前端与数据库的中间人, 他会将前端想要的数据从数据库中读出来给前端。他也会将用户要想保存的数据写到数据库。

区别:

MVC 中的 View 的目的是「呈现哪一个数据」, 而 MTV 的 View 的目的是「数据如何呈现」。

也就是把 MVC 中的 View 分成了视图 (展现哪些数据) 和模板 (如何展现) 2 个部分, 而 Controller 这个要素由框架自己来实现了, 我们需要做的就是 (带正则表达式的) URL 对应到视图就可以了, 通过这样的 URL 配置, 系统将一个请求发送到一个合适的视图。设计模式:

是一套被反复使用, 多数人知道并经过分类的代码设计经验总结, 是为了解决一些通用性问题的

目的: 重用代码并保证代码的可靠性

设计模式分类: 单例, 抽象工厂 等等 23 种模式

一句话总结: 解决某一些特殊问题的一种思想和思路

框架模式:

代码重用, 框架模式是解决如何设计程序框架的代码, 在框架模式中会包含多种设计模式, 与设计模式是一种包含关系,

举例来说: 比如要盖楼, 那怎么盖楼属于框架模式, 楼里面的电梯怎么设计, 楼梯怎么设计, 属于设计模式, 所以框架模式在盖楼中属于如何把楼盖起来, 那么他里面会包含多种设计模式, 具体的细节碰到不同的东西, 会采用不同的设计模式来解决, 因此在一种框架模式中会包含多种设计模式。

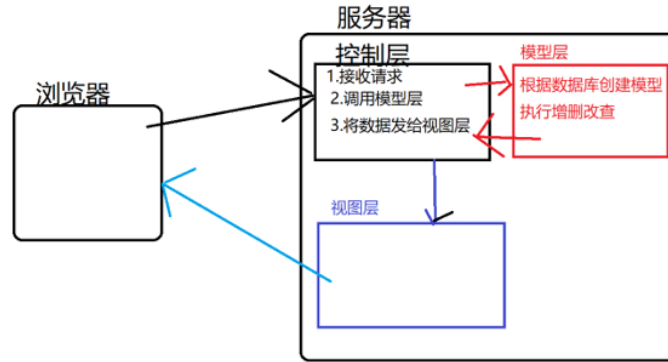
目前流行的框架模式:

MVC(适用于多种编程语言，单在 python 中不常用):

M: Models 模型层，在程序中用于处理数据逻辑的部分，（主要是处理数据），主要负责在数据库中对数据的存取操作，其实就是对数据库的增删改查的操作

V: Views 视图层，在应用程序中处理显示部分的内容（html,jsp）

C: Controllers 控制层，处理用户交互的部分，主要作用于M和V之间做协调，通常是负责从模型层中抽取数据，再进行业务处理，最后将数据传给视图层，并将视图传给客户端



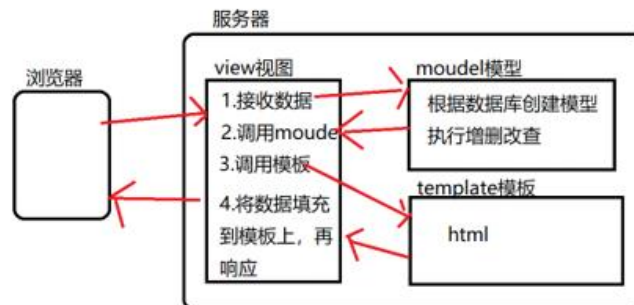
详解：用户首先打开浏览器，输入网址，然后浏览器向服务器发送请求，到了服务器之后，由控制层接收这个请求，接收完请求就知道用户想要做什么，了解了用户的意图，如果需要用到一些数据，比如想查看某某商品的信息，那么控制器就需要找到商品的信息，所以控制器就找模型层了，模型层会根据数据库创建模型（注意模型层不是数据库）一般情况数据库有多少张表，那么模型层就有多少个类，每个表中有多少个字段，模型层中的类就有多少个变（属性），在模型层里还会提供增删改查的操作，那么这个执行结构再反馈给控制器，到此，控制层和模型层的交互完成了，接下来，控制层就会把接收到的数据发送给视图，视图会把数据显示在网页里，反馈给浏览器，这样用户就看到了

MTV(django)

M: 模型层，功能同上

T: templates:模板层，用于处理用户显示部分的内容，和 MVC 中的 V 是一样的，通过 html 展示

V: views 视图层，在 MTV 中视图层是处理用户交互的部分，从模型层中获取数据，再将数据交给模板层，再先是给用



详解：用户打开浏览器，浏览器发送请求，视图层接收用户请求，接受完请求调用模型层，模型层根据数据库创建模型，进行增删改查等操作，模型层处理完数据返回给视图层，视图层接收完数据调用模板层，模板层里存放 HTML 等页面，模板层会把 HTML 模板页面返回给视图层，视图层填充数据到模板上，然后再返回给浏览器

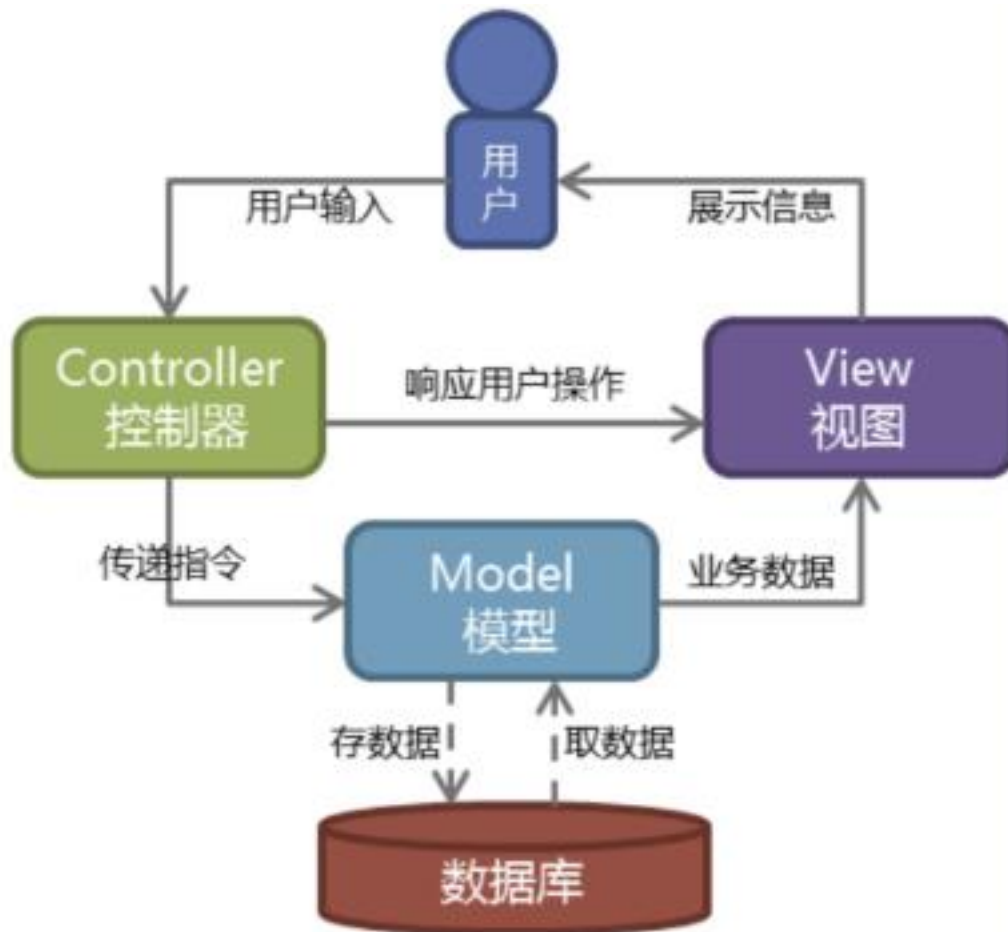
MVC 框架

Web 服务器开发领域里著名的 MVC 模式，所谓 MVC 就是把 Web 应用分为模型(M)，控制器(C)和视图(V)三层，他们之间以一种插件式的、松耦合的方式连接在一起，模型负责业务对象与数据库的映射(ORM)，视图负责与用户的交互(页面)，控制器接受用户的输入调用模型和视图完成用户的请求，其示意图如下所示：

M -- models 数据库相关

V -- views 视图相关(逻辑)

C -- controller url 控制器(url 分发器,路由分发)



MTV 框架

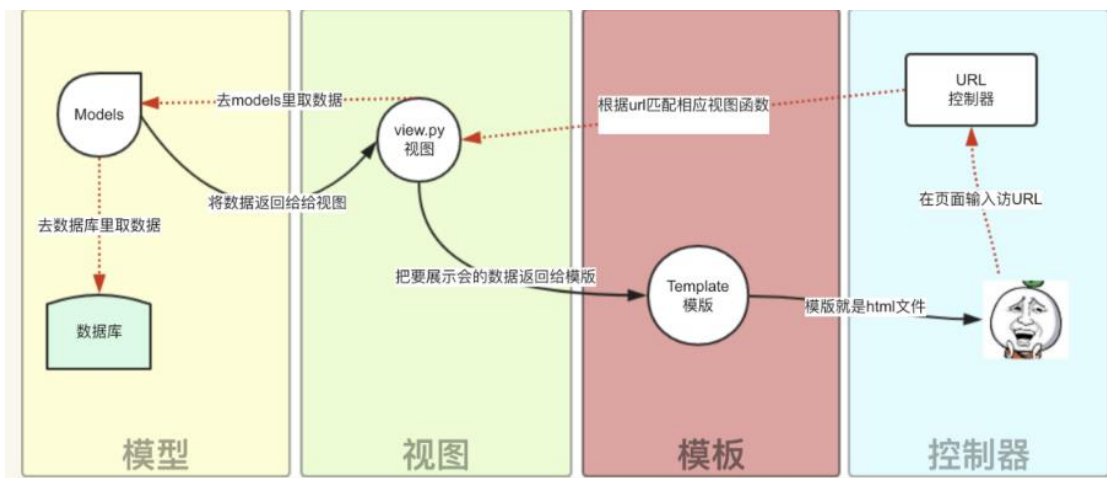
Django 的 MTV 模式本质上和 MVC 是一样的，也是为了各组件间保持松耦合关系，只是定义上有些许不同，Django 的 MTV 分别是值：

M 代表模型 (Model)：负责业务对象和数据库的关系映射(ORM)。

T 代表模板 (Template)：负责如何把页面展示给用户(html)。

V 代表视图 (View)：负责业务逻辑，并在适当时候调用 Model 和 Template。

除了以上三层之外，还需要一个 URL 分发器，它的作用是将一个个 URL 的页面请求分发给不同的 View 处理，View 再调用相应的 Model 和 Template，MTV 的响应模式如下所示：



2. Django 流程,中间件的执行流程以及如何自定义中间件。

Python 的 WEB 框架有 Django、Tornado、Flask、Zope TurboGears、Web2py(Webpy)、Pylons 等多种，Django 相较于其他 WEB 框架其优势为：大而全，框架本身集成了 ORM、模型绑定、模板引擎、缓存、Session 等诸多功能。

流程

基本配置

路由系统

视图 view

模板

Model

中间件

Form

认证系统

CSRF

分页

Cookie

Seesion

缓存

序列化

信号

admin

一、Django 中间件的运行流程

- django 的生命周期是：前端请求--->nginx--->uwsgi--->中间件--->url 路由---->view 试图--->orm---->拿到数据返回给 view---->试图将数据渲染到模版中拿到字符串---->中间件--->uwsgi---->nginx---->前端渲染。
- Django 中的中间件是一个轻量级、底层的插件系统，可以介入 Django 的请求和响应处理过程，修改 Django 的输入或输出。中间件的设计为开发者提供了一种无侵入式的开发方式，增强了 Django 框架的健壮性。
- 我们可以使用中间件，在 Django 处理视图的不同阶段对输入或输出进行干预。

二、中间件的定义方法

- 定义一个中间件工厂函数，然后返回一个可以别调用的中间件。
- 中间件工厂函数需要接收一个可以调用的 `get_response` 对象。

- 返回的中间件也是一个可以被调用的对象，并且像视图一样需要接收一个 `request` 对象参数，返回一个 `response` 对象。
- 中间件的示例如下：

```
def middleware(get_response):
    # 此处编写的代码仅在 Django 第一次配置和初始化的时候执行一次。
    def middleware(request):
        # 此处编写的代码会在每个请求处理视图前被调用。
        response = get_response(request)
        # 此处编写的代码会在每个请求处理视图之后被调用。
        return response
    return middleware
```

- 例如，在 `users` 应用中新建一个 `middleware.py` 文件

```
def my_middleware(get_response):
    print('init 被调用')
    def middleware(request):
        print('before request 被调用')
        response = get_response(request)
        print('after response 被调用')
        return response
    return middleware
```

`'users.middleware.my_middleware', # 添加中间件`

- 定义一个视图函数进行测试在 `users.py` 文件下进行定义

```
def demo_view(request):
    print('view 视图被调用')
    return HttpResponse('OK')
```

三、多个中间件的执行顺序

以下是在项目主目录下 `settings.py` 文件下进行

- 在请求视图被处理前，中间件由上至下依次执行
- 在请求视图被处理后，中间件由下至上依次执行

举例：重新定义一个中间件

```
def my_middleware2(get_response):
    print('init2 被调用')
    def middleware(request):
        print('before request 2 被调用')
        response = get_response(request)
        print('after response 2 被调用')
        return response
    return middleware
```

四、知识点补充

1、如果开启 `django` 的时候端口被占用，可用下面的方式来查看端口的情况。

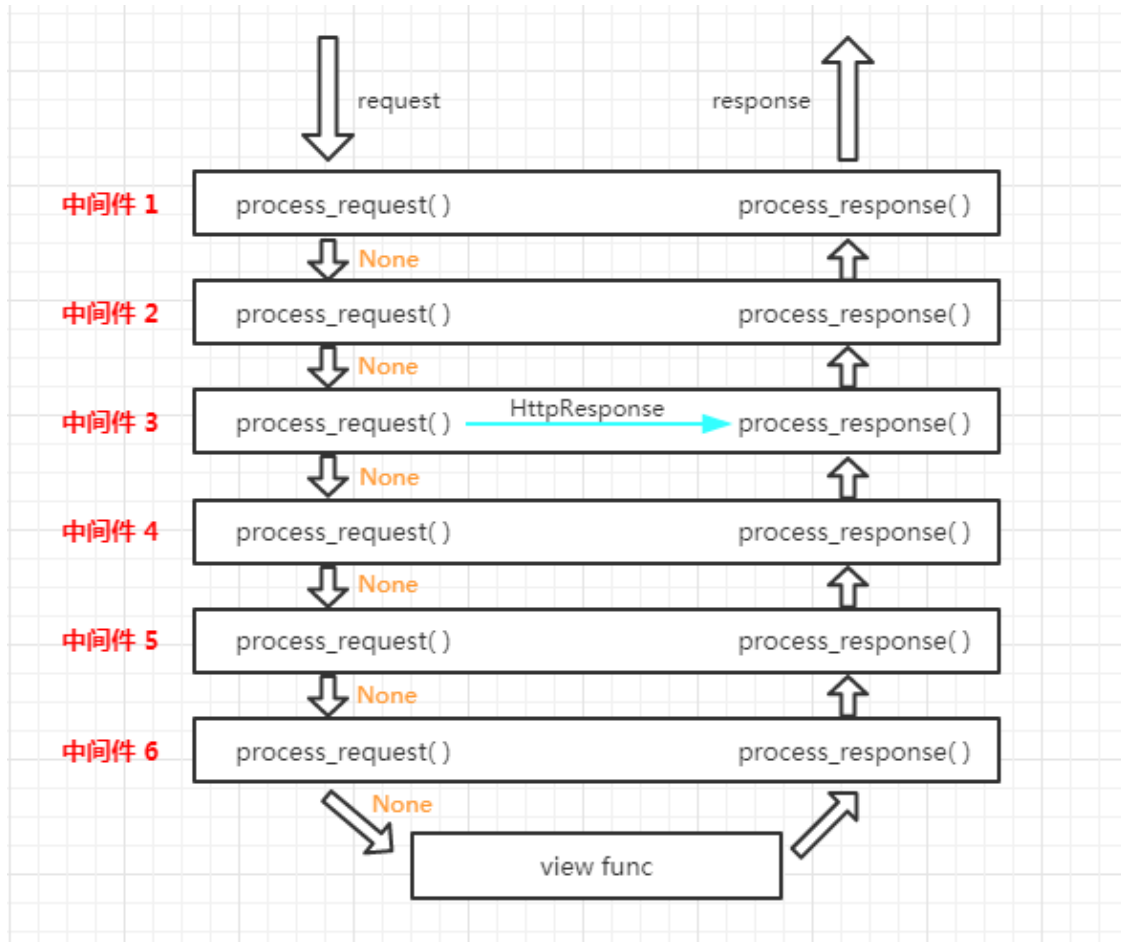
ps -e | grep python

2、杀死被占用的端口

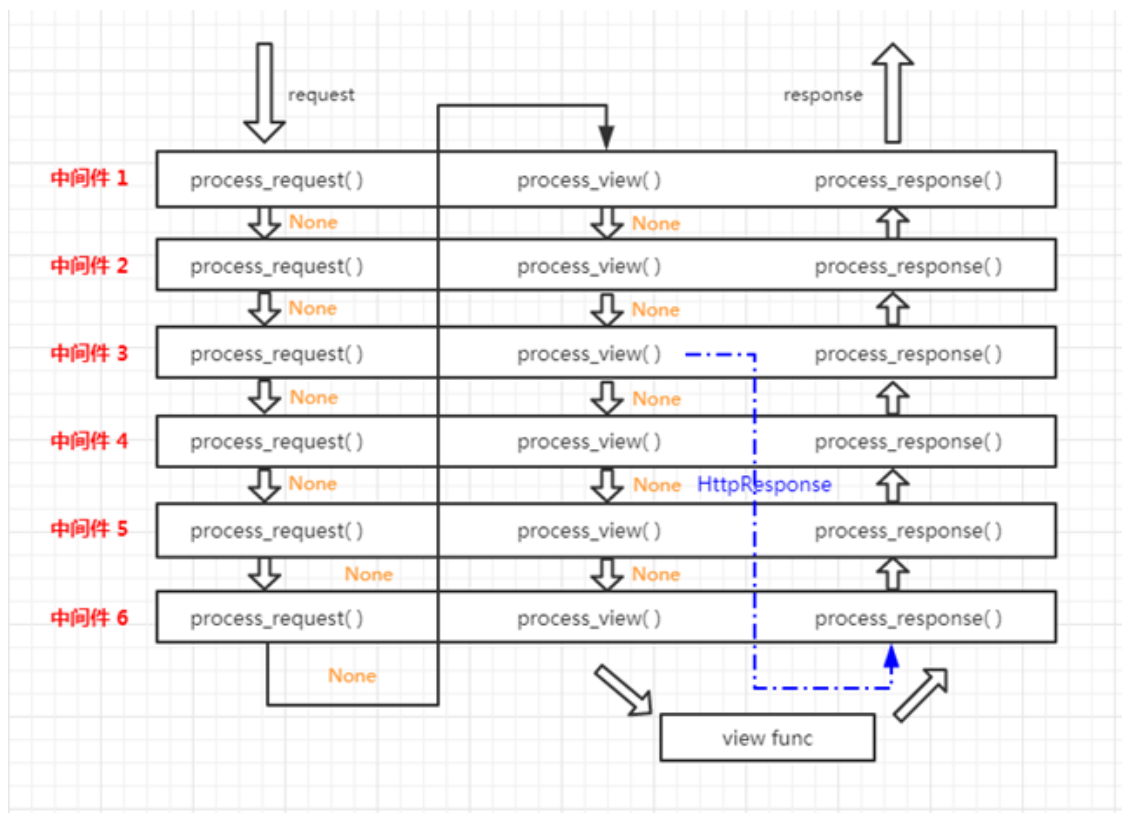
kill -9 端口号

中间件的执行流程

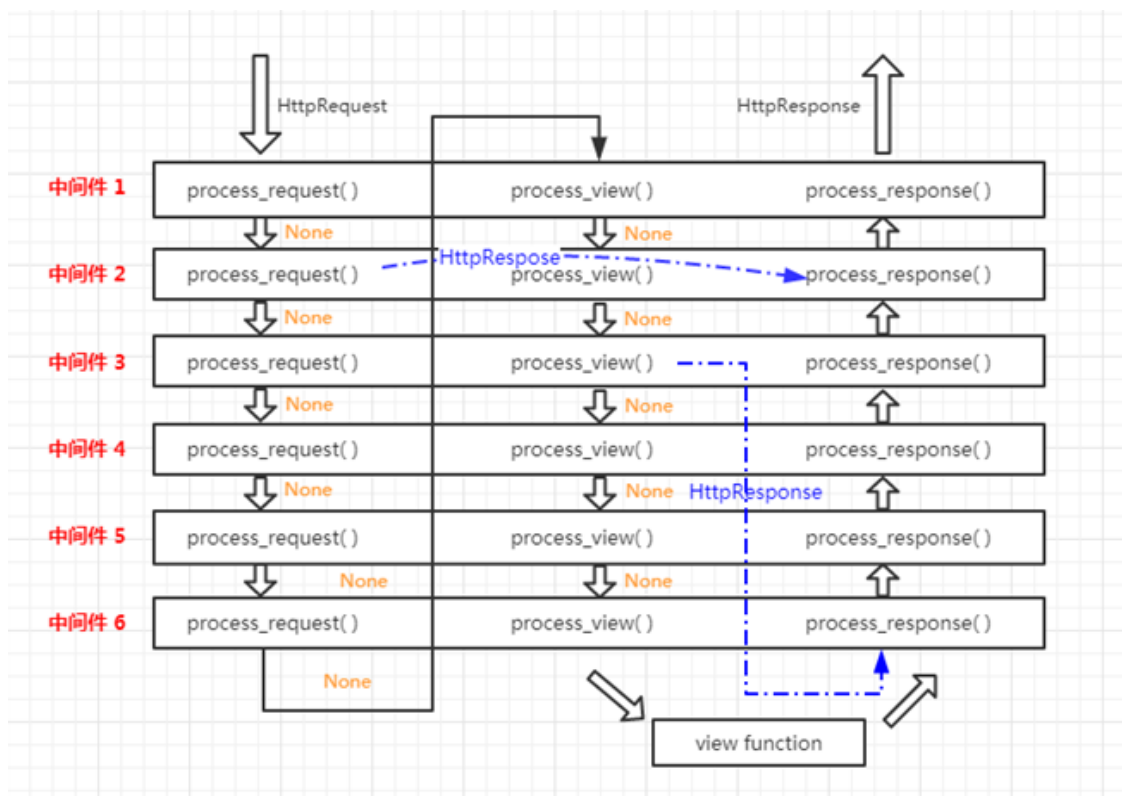
1、请求到达中间件之后，先按照正序执行每个注册中间件的 `processrequest` 方法，`processrequest` 方法返回的值是 `None`，就依次执行，如果返回的值是 `HttpResponse` 对象，不再执行后面的 `processrequest` 方法，而是执行当前对应中间件的 `processresponse` 方法，将 `HttpResponse` 对象返回给浏览器。（也就是说：如果 MIDDLEWARE 中注册了 6 个中间件，执行过程中，第 3 个中间件返回了一个 `HttpResponse` 对象，那么第 4,5,6 中间件的 `processrequest` 和 `processresponse` 方法都不执行，顺序执行 3,2,1 中间件的 `process_response` 方法。）

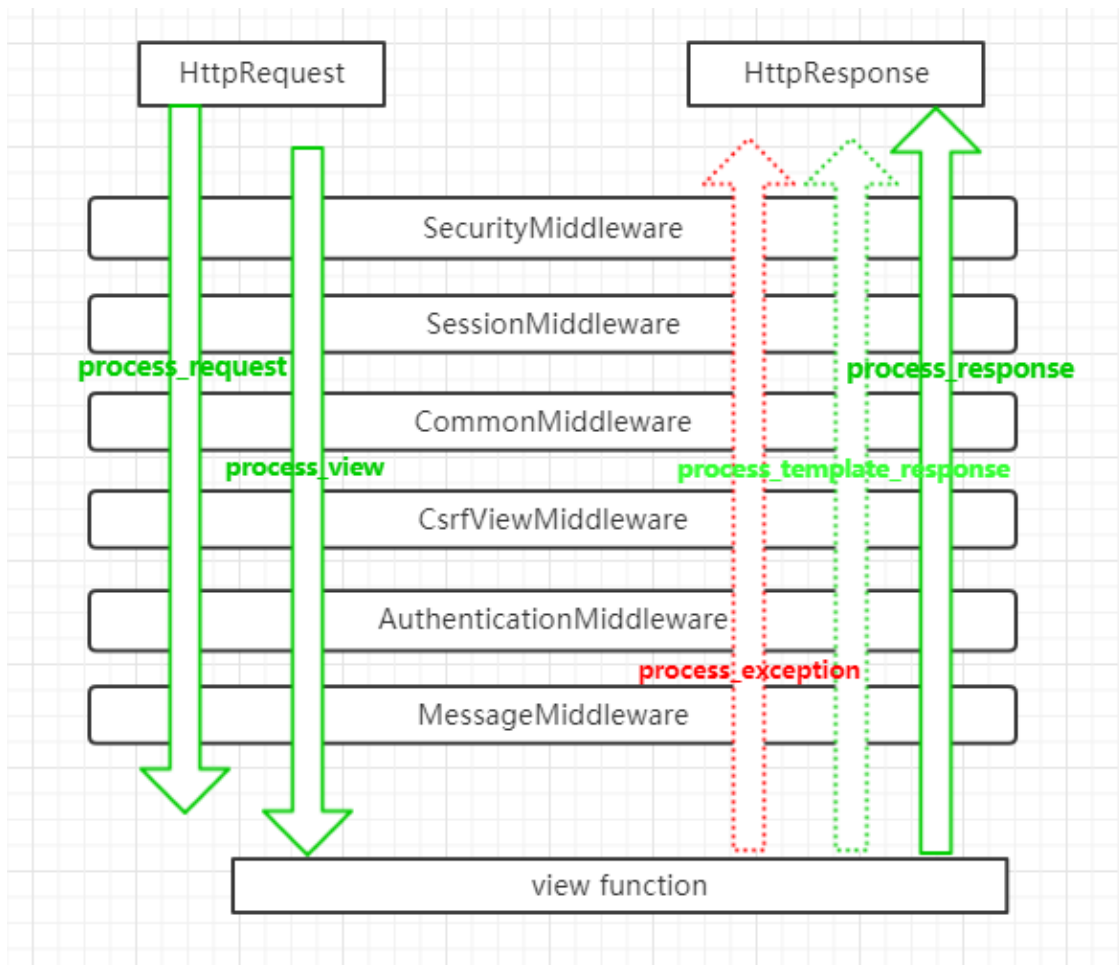


2、`processrequest` 方法都执行完后，匹配路由，找到要执行的视图函数，先不执行视图函数，先执行中间件中的 `processview` 方法，`processview` 方法返回 `None`，继续按顺序执行，所有 `processview` 方法执行完后执行视图函数。（加入中间件 3 的 `processview` 方法返回了 `HttpResponse` 对象，则 4,5,6 的 `processview` 以及视图函数都不执行，直接从最后一个中间件，也就是中间件 6 的 `process_response` 方法开始倒序执行。）



3、`process_template_response` 和 `process_exception` 两个方法的触发是有条件的，执行顺序也是倒序。总结所有的执行流程如下：





简单的 session 认证

注意~如果是基于 session 的认证，需要写在 Django 自带的 session 中间件的下面！

```
class M1(MiddlewareMixin):
    def process_request(self, request):
        # 设置路径白名单，只要访问的是 login 登陆路径，就不做这个 cookie 认证
        if request.path not in [reverse('login'),]:
            print('我是 M1 中间件') # 客户端 IP 地址
            # return HttpResponse('sorry, 没有通过我的 M1 中间件')
            is_login = request.COOKIES.get('is_login', False)

            if is_login:
                pass
            else:
                # return render(request, 'login.html')
                return redirect(reverse('login'))
        else:
            return None # 别忘了 return None，或者直接写个 pass
    def process_response(self, request, response):
        print('M1 响应部分')
        # print(response.__dict__['_container'][0].decode('utf-8'))
        return response
        # return HttpResponse('认证失败')
```

中间件版登陆认证程序--用到白名单

中间件版的登录验证需要依靠 session，所以数据库中要有 django_session 表。

路由

```
from django.conf.urls import url
from app01 import views

urlpatterns = [
    url(r'^index/$', views.index),
    url(r'^login/$', views.login, name='login'),
]
```

视图函数

```
from django.shortcuts import render, HttpResponseRedirect, redirect

def index(request):
    return HttpResponseRedirect('this is index')
def home(request):
    return HttpResponseRedirect('this is home')

def login(request):
    if request.method == "POST":
        user = request.POST.get("user")
        pwd = request.POST.get("pwd")

        if user == "Q1mi" and pwd == "123456":
            # 设置 session
            request.session["user"] = user
            # 获取跳到登陆页面之前的 URL
            next_url = request.GET.get("next")
            # 如果有，就跳转回登陆之前的 URL
            if next_url:
                return redirect(next_url)
            # 否则默认跳转到 index 页面
            else:
                return redirect("/index/")
        return render(request, "login.html")
```

login 页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>登录页面</title>
</head>
<body>
<form action="{% url 'login' %}" method="post">
    <p>
        <label for="user">用户名: </label>
        <input type="text" name="user" id="user">
    </p>
    <p>
        <label for="pwd">密 码: </label>
```



```

        <input type="text" name="pwd" id="pwd">
    </p>
    <input type="submit" value="登录">
</form>
</body>
</html>

```

自定义中间件的内容

自定义中间件中 `middlewares.py` 的内容如下：

```

class AuthMD(MiddlewareMixin):
    white_list = ['/login/', ] # 白名单
    balck_list = ['/black/', ] # 黑名单
    def process_request(self, request):
        from django.shortcuts import redirect, HttpResponseRedirect
        #先拿到用户想要访问的页面，登陆成功后跳转到用户想要去的那个页面
        next_url = request.path_info
        print(request.path_info, request.get_full_path()) #注意 path_info 与 get_full_path()的区别

        if next_url in self.white_list or request.session.get("user"):
            return None
        elif next_url in self.balck_list: #如果在黑名单的路径
            return HttpResponseRedirect('This is an illegal URL')
        else:
            #登陆成功后跳转到用户想要去的那个页面
            return redirect("/login/?next={}".format(next_url))

```

在 `settings.py` 中注册

我在项目中新建了一个 `core` 包，`middlewares.py` 文件写在了这个包里：

```

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'core.middlewares.AuthMD',
]

```

`AuthMD` 中间件注册后，所有的请求都要走 `AuthMD` 的 `process_request` 方法。

访问的 URL 在白名单内或者 `session` 中有 `user` 用户名，则不做阻拦走正常流程；

如果 URL 在黑名单中，则返回 `This is an illegal URL` 的字符串；

正常的 URL 但是需要登录后访问，让浏览器跳转到登录页面。

注：`AuthMD` 中间件中需要 `session`，所以 `AuthMD` 注册的位置要在 `session` 中间的下方。

[回到顶部](#)

DRF 中的频率组件

可以限制用户访问的频率

在 throttles.py 中

```
from rest_framework.throttling import BaseThrottle, SimpleRateThrottle
import time
from rest_framework import exceptions
visit_record = {}
class VisitThrottle(BaseThrottle):
    # 限制访问时间
    VISIT_TIME = 10
    VISIT_COUNT = 3
    # 定义方法 方法名和参数不能变
    def allow_request(self, request, view):
        # 获取登录主机的 id
        id = request.META.get('REMOTE_ADDR')
        self.now = time.time()
        if id not in visit_record:
            visit_record[id] = []
        self.history = visit_record[id]
        # 限制访问时间
        while self.history and self.now - self.history[-1] > self.VISIT_TIME:
            self.history.pop()
        # 此时 history 中只保存了最近 10 秒钟的访问记录
        if len(self.history) >= self.VISIT_COUNT:
            return False
        else:
            self.history.insert(0, self.now)
            return True
    def wait(self):
        return self.history[-1] + self.VISIT_TIME - self.now
```

在 views.py 中

```
from app01.service.throttles import *

class BookViewSet(generics.ListCreateAPIView):
    throttle_classes = [VisitThrottle,]
    queryset = Book.objects.all()
    serializer_class = BookSerializers
```

全局视图 throttle

```
REST_FRAMEWORK={
    "DEFAULT_AUTHENTICATION_CLASSES":["app01.service.auth.Authentication",],
    "DEFAULT_PERMISSION_CLASSES":["app01.service.permissions.SVIPPermission",],
    "DEFAULT_THROTTLE_CLASSES":["app01.service.throttles.VisitThrottle",]
}
```

内置 throttle 类

在 throttles.py 修改为:

```
class VisitThrottle(SimpleRateThrottle):
    scope="visit_rate"
    def get_cache_key(self, request, view):
        return self.get_ident(request)
```

settings.py 设置

```
REST_FRAMEWORK={
    "DEFAULT_AUTHENTICATION_CLASSES":["app01.service.auth.Authentication",],
```

```
"DEFAULT_PERMISSION_CLASSES":["app01.service.permissions.SVIPPermission",],
"DEFAULT_THROTTLE_CLASSES":["app01.service.throttles.VisitThrottle",],
"DEFAULT_THROTTLE_RATES":{
    "visit_rate":"5/m",
}
}
```

Django 基础九之中间件

本节目录

- 一 前戏
- 二 中间件介绍
- 三 自定义中间件
- 四 中间件的执行流程
- 五 中间件版登陆认证
- 六 xxx
- 七 xxx
- 八 xxx
-

一 前戏

我们在前面的课程中已经学会了给视图函数加装饰器来判断是用户是否登录，把没有登录的用户请求跳转到登录页面。我们通过给几个特定视图函数加装饰器实现了这个需求。但是以后添加的视图函数可能也需要加上装饰器，这样是不是稍微有点繁琐。

学完今天的内容之后呢，我们就可以用更适宜的方式来实现类似给所有请求都做相同操作的功能了

二 中间件介绍

中间件顾名思义，是介于 **request** 与 **response** 处理之间的一道处理过程，相对比较轻量级，并且在全局上改变 **django** 的输入与输出。因为改变的是全局，所以需要谨慎实用，用不好会影响到性能。

Django 的中间件的定义：

Middleware is a framework of hooks into Django's request/response processing.
It's a light, low-level "plugin" system for globally altering Django's input or output.

如果你想修改请求，例如被传送到 **view** 中的 **HttpRequest** 对象。或者你想修改 **view** 返回的 **HttpResponse** 对象，这些都可以通过中间件来实现。

可能你还想在 **view** 执行之前做一些操作，这种情况就可以用 **middleware** 来实现。

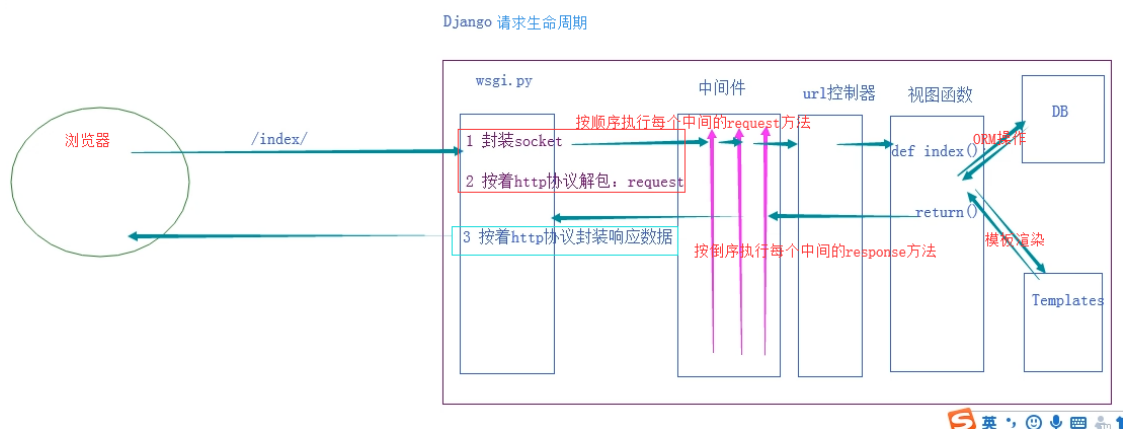
说的直白一点中间件是帮助我们在视图函数执行之前和执行之后都可以做一些额外的操作，它本质上就是一个自定义类，类中定义了几个方法，Django 框架会在请求的特定的时间去执行这些方法。

我们一直都在使用中间件，只是没有注意到而已，打开 Django 项目的 Settings.py 文件，看到下面的 MIDDLEWARE 配置项，django 默认自带的一些中间件：

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

MIDDLEWARE 配置项是一个列表，列表中是一个个字符串，这些字符串其实是一个个类，也就是一个个中间件。

我们之前已经接触过一个 csrf 相关的中间件了？我们一开始让大家把他注释掉，再提交 post 请求的时候，就不会被 forbidden 了，后来学会使用 csrf_token 之后就不再注释这个中间件了。



那接下来就学习中间件中的方法以及这些方法什么时候被执行。

三 自定义中间件

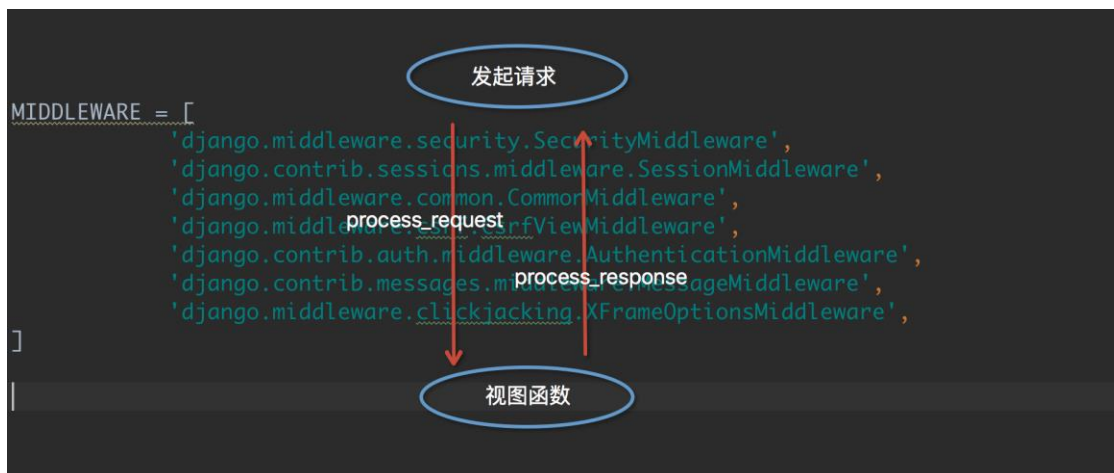
想了解更多中间件，在[开源中国](#)中有很多关于中间件的[详细解释](#)

中间件可以定义五个方法，分别是：（主要的是 `processrequest` 和 `processresponse`）

- `process_request(self,request)`
- `processview(self, request, viewfunc, viewargs, viewkwargs)`
- `processtemplateresponse(self,request,response)`
- `process_exception(self, request, exception)`
- `process_response(self, request, response)`

以上方法的返回值可以是 `None` 或一个 `HttpResponse` 对象，如果是 `None`，则继续按照 django 定义的规则向后继续执行，如果是 `HttpResponse` 对象，则直接将该对象返回给用户。

当用户发起请求的时候会依次经过所有的的中间件，这个时候的请求时 `processrequest`,最后到达 `views` 的函数中，`views` 函数处理后，在依次穿过中间件，这个时候是 `processresponse`,最后返回给请求者。



上述截图中的中间件都是 `django` 中的，我们也可以自己定义一个中间件，我们可以自己写一个类，但是必须继承 `MiddlewareMixin`

自定义一个中间件示例

目录：

在项目中创建一个包，随便起名字，一般都放在一个叫做 `utils` 的包里面，表示一个公用的组件，创建一个 `py` 文件，随便起名字，例如叫做：`middlewares.py`，内容如下

```
from django.utils.deprecation import MiddlewareMixin  
class MD1(MiddlewareMixin):  
    #自定义中间件，不是必须要有下面这两个方法，有 request 方法说明请求来了要处理，有 response 方法说明响应出去时需要处理，不是非要写这两个方法，如果你没写 process_response 方法，那么会一层一层的往上找，哪个中间件有 process_response 方法就将返回对象给哪个中间件  
    def process_request(self, request):  
        print("MD1 里面的 process_request")  
  
    def process_response(self, request, response):  
        print("MD1 里面的 process_response")  
        return response
```

process_request

`process_request` 有一个参数，就是 `request`，这个 `request` 和视图函数中的 `request` 是一样的。

它的返回值可以是 `None` 也可以是 `HttpResponse` 对象。返回值是 `None` 的话，按正常流程继续走，交给下一个中间件处理，如果是 `HttpResponse` 对象，`Django` 将不执行视图函数，而将相应对象返回给浏览器。

我们来看看多个中间件时，`Django` 是如何执行其中的 `process_request` 方法的。

```
from django.utils.deprecation import MiddlewareMixin  
class MD1(MiddlewareMixin):  
    def process_request(self, request):  
        print("MD1 里面的 process_request")  
class MD2(MiddlewareMixin):
```

```
def process_request(self, request):
    print("MD2 里面的 process_request")
    pass
```

在 settings.py 的 MIDDLEWARE 配置项中注册上述两个自定义中间件：

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'middlewares.MD1', # 自定义中间件 MD1, 这个写的是你项目路径下的一个路径, 例如, 如果你放在
    # 项目下, 文件夹名成为 utils, 那么这里应该写 utils.middlewares.MD1
    'middlewares.MD2' # 自定义中间件 MD2
]
```

此时, 我们访问一个视图, 会发现终端中打印如下内容:

```
MD1 里面的 process_request
MD2 里面的 process_requestapp01 中的 index 视图
```

把 MD1 和 MD2 的位置调换一下, 再访问一个视图, 会发现终端中打印的内容如下:

```
MD2 里面的 process_request
MD1 里面的 process_requestapp01 中的 index 视图
```

看结果我们知道: 视图函数还是最后执行的, MD2 比 MD1 先执行自己的 process_request 方法。

在打印一下两个自定义中间件中 process_request 方法中的 request 参数, 会发现它们是同一个对象。

由此总结一下:

中间件的 process_request 方法是在执行视图函数之前执行的。

当配置多个中间件时, 会按照 MIDDLEWARE 中的注册顺序, 也就是列表的索引值, 从前到后依次执行的。

不同中间件之间传递的 request 都是同一个对象

多个中间件中的 process_response 方法是按照 MIDDLEWARE 中的注册顺序倒序执行的, 也就是说第一个中间件的 processrequest 方法首先执行, 而它的 processresponse 方法最后执行, 最后一个中间件的 processrequest 方法最后一个执行, 它的 process_response 方法是最先执行。

process_response

它有两个参数, 一个是 request, 一个是 response, request 就是上述例子中一样的对象, response 是视图函数返回的 HttpResponse 对象。该方法的返回值也必须是 HttpResponse 对象。

给上述的 M1 和 M2 加上 process_response 方法:

```
from django.utils.deprecation import MiddlewareMixin
class MD1(MiddlewareMixin):
```

```

def process_request(self, request):
    print("MD1 里面的 process_request")
    #不必写 return 值
def process_response(self, request, response):#request 和 response 两个参数必须有，名字随便取
    print("MD1 里面的 process_response")          #print(response.__dict__['_container
']][0].decode('utf-8')) #查看响应体里面的内容的方法
    return response #必须有返回值，写 return response ，这个 response 就像一个接力棒一样

    #return HttpResponseRedirect('瞎搞')，如果你写了这个，那么你视图返回过来的内容就被它给替代了
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2 里面的 process_request")
        pass
    def process_response(self, request, response): #request 和 response 两个参数必须要有，名字随便取
        print("MD2 里面的 process_response")
        return response #必须返回 response，不然你上层的中间件就没有拿到 httpresponse 对象，就会报错

```

访问一个视图，看一下终端的输出：

```

MD2 里面的 process_request
MD1 里面的 process_request
app01 中的 index 视图
MD1 里面的 process_response
MD2 里面的 process_response

```

看结果可知：

`process_response` 方法是在视图函数之后执行的，并且顺序是 MD1 比 MD2 先执行。（此时 `settings.py` 中 MD2 比 MD1 先注册）

多个中间件中的 `processresponse` 方法是按照 `MIDDLEWARE` 中的注册顺序倒序执行的，也就是说第一个中间件的 `processrequest` 方法首先执行，而它的 `processresponse` 方法最后执行，最后一个中间件的 `processrequest` 方法最后一个执行，它的 `process_response` 方法是最先执行。

再看一个例子：

```

from django.utils.deprecation import MiddlewareMixin
from django.shortcuts import HttpResponseRedirect
class Md1(MiddlewareMixin):
    def process_request(self,request):
        print("Md1 请求")          #process_request 方法里面不写返回值，默认也是返回 None，如果你自己写了 return None，也是一样的效果，不会中断你的请求，但是如果你 return 的一个 httpresponse 对象，那么就会在这个方法中断你的请求，直接返回给用户，这就成了非正常的流程了          #并且，如果你在这里 return 了 httpresponse 对象，那么会从你这个中间件类中的 process_response 方法开始执行返回操作，所以这个类里面只要有 process_response 方法，肯定会执行

    def process_response(self,request,response):
        print("Md1 返回")
        return response
class Md2(MiddlewareMixin):

```

```
def process_request(self,request):
    print("Md2 请求")
    #return HttpResponse("Md2 中断")
def process_response(self,request,response):
    print("Md2 返回")
    return response
```

结果:

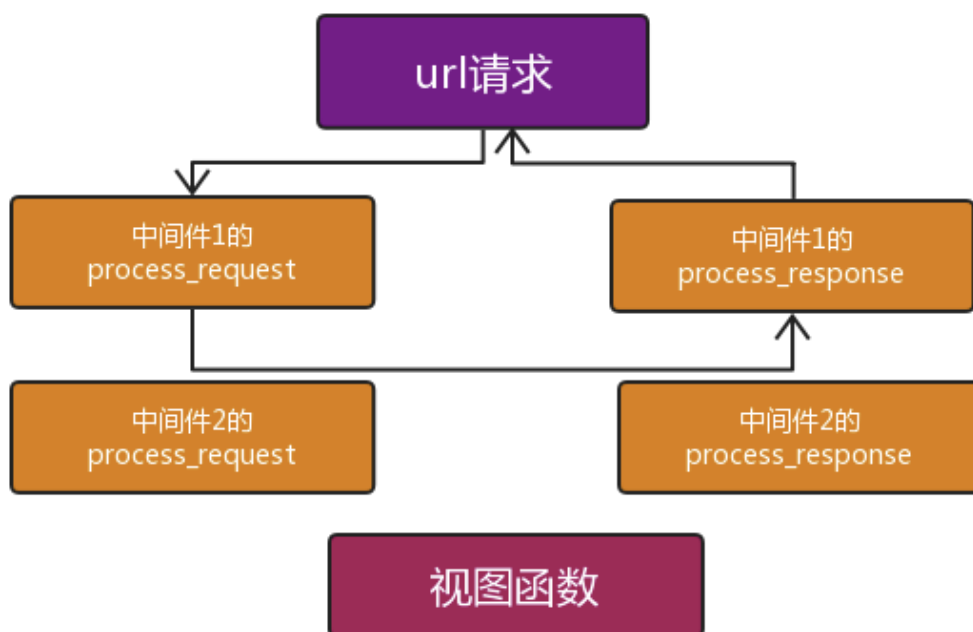
Md1 请求
Md2 请求
view 函数...
Md2 返回
Md1 返回

注意: 如果当请求到达请求 2 的时候直接不符合条件返回, 即 **return HttpResponse("Md2 中断")**, 程序将把请求直接发给中间件 2 返回, 然后依次返回到请求者, 结果如下:

返回 Md2 中断的页面, 后台打印如下:

Md1 请求
Md2 请求
Md2 返回
Md1 返回

流程图如下:



之前我们做的 cookie 认证, 都是通过在函数上面加装饰器搞的, 比较麻烦, 看看中间件怎么搞, 如果写的是 session 认证的, 你必须放在 django 自带的 session 中间件的下面, 所以自定义中间之后, 你需要注意你的中间件的摆放顺序。

```

class M1(MiddlewareMixin):
    def process_request(self,request):
        #设置路径白名单，只要访问的是 login 登陆路径，就不做这个 cookie 认证
        if request.path not in [reverse('login'),]:
            print('我是 M1 中间件') #客户端 IP 地址
            # return HttpResponse('sorry,没有通过我的 M1 中间件')
            is_login = request.COOKIES.get('is_login', False)
            if is_login:
                pass
            else:
                # return render(request,'login.html')
                return redirect(reverse('login'))
        else:
            return None #别忘了 return None
    def process_response(self,request,response):
        print('M1 响应部分')
        # print(response.__dict__['_container'][0].decode('utf-8'))
        return response
        # return HttpResponse('瞎搞')

```

练习：尝试一下通过中间件来控制用户的访问次数，让用户在一分钟之内不能访问我的网站超过 20 次。

后面要学的方法不常用，但是大家最好也要知道。

process_view

`processview(self, request, viewfunc, viewargs, viewkwargs)`

该方法有四个参数

`request` 是 `HttpRequest` 对象。

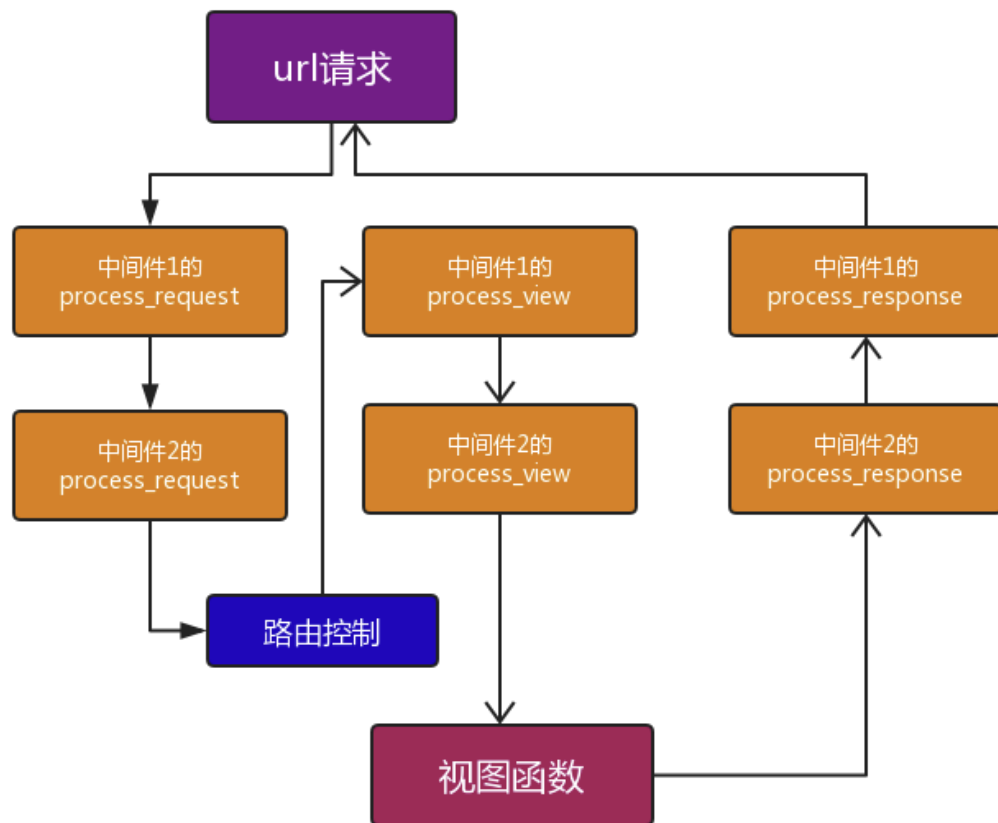
`view_func` 是 Django 即将使用的视图函数。（它是实际的函数对象，而不是函数的名称作为字符串。）

`view_args` 是将传递给视图的位置参数的列表。

`viewkwargs` 是将传递给视图的关键字参数的字典。`viewargs` 和 `view_kwargs` 都不包含第一个视图参数（`request`）。

Django 会在调用视图函数之前调用 `process_view` 方法。

它应该返回 `None` 或一个 `HttpResponse` 对象。如果返回 `None`，Django 将继续处理这个请求，执行任何其他中间件的 `processview` 方法，然后在执行相应的视图。如果它返回一个 `HttpResponse` 对象，Django 不会调用对应的视图函数。它将执行中间件的 `processresponse` 方法并将应用到该 `HttpResponse` 并返回结果。



给 MD1 和 MD2 添加 process_view 方法:

```

from django.utils.deprecation import MiddlewareMixin
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1 里面的 process_request")
    def process_response(self, request, response):
        print("MD1 里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)
        print("MD1 中的 process_view")
        print(view_func, view_func.__name__) #就是 url 映射到的那个视图函数，也就是说每个中
        #间件的这个 process_view 已经提前拿到了要执行的那个视图函数 #ret = view_func(request)
        #提前执行视图函数，不用到了上图的视图函数的位置再执行，如果你视图函数有参数的话，可以这么写 view
        _func(request,view_args,view_kwargs)
        #return ret #直接就在 MD1 中间件这里这个类的 process_response 给返回了，就不会去找到
        #视图函数里面的这个函数去执行了。
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2 里面的 process_request")
        pass
    def process_response(self, request, response):
        print("MD2 里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)

```

```
print("MD2 中的 process_view")
print(view_func, view_func.__name__)
```

访问 index 视图函数，看一下输出结果：

MD2 里面的 process_request

MD1 里面的 process_request

MD2 中的 process_view

<function index at 0x000001DE68317488> index

MD1 中的 process_view

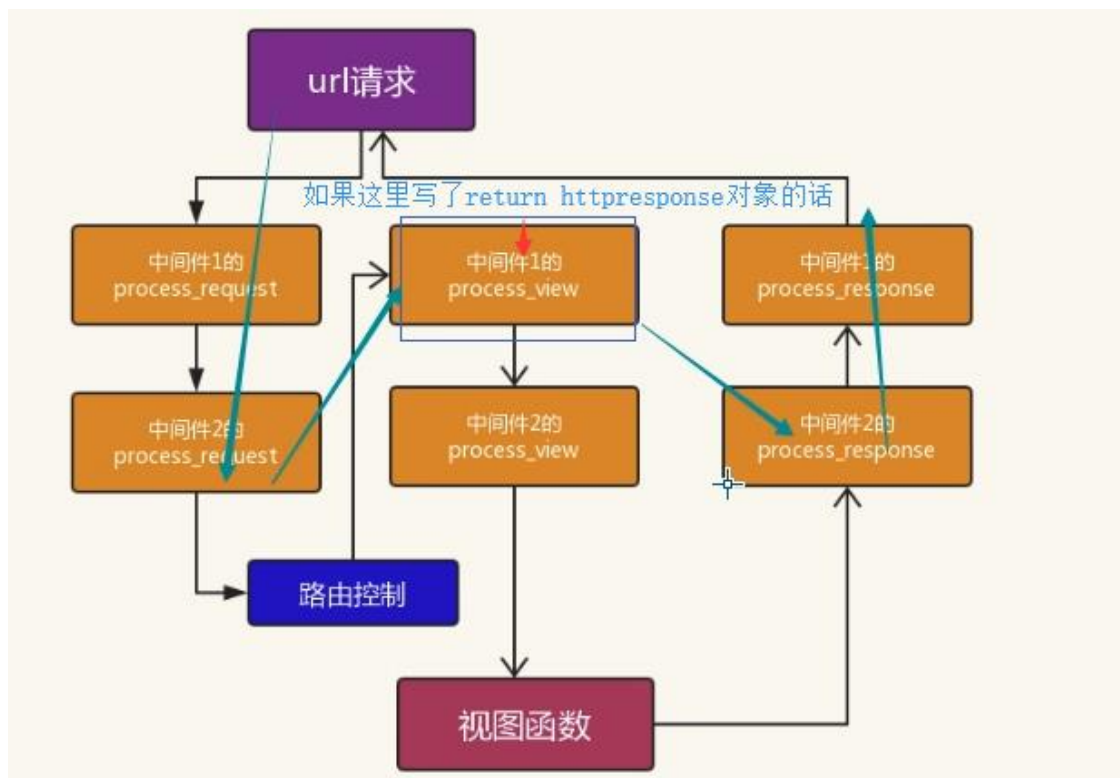
<function index at 0x000001DE68317488> index

app01 中的 index 视图

MD1 里面的 process_response

MD2 里面的 process_response

processview 方法是在 processrequest 之后，reprocess_response 之前，视图函数之前执行的，执行顺序按照 MIDDLEWARE 中的注册顺序从前到后顺序执行的



process_exception

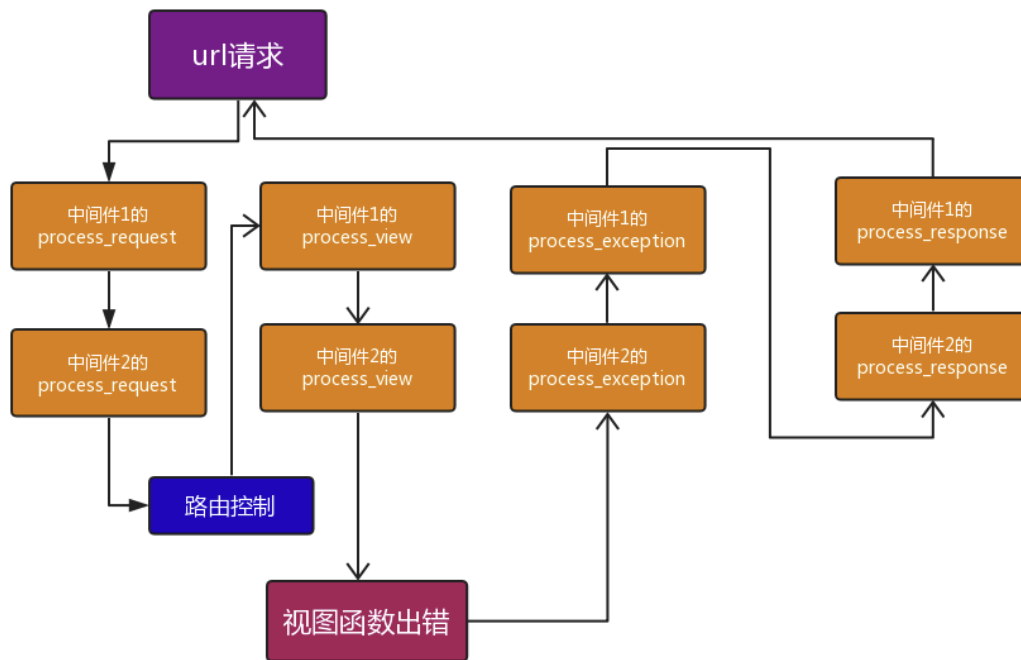
process_exception(self, request, exception)

该方法两个参数：

一个 HttpRequest 对象

一个 exception 是视图函数异常产生的 Exception 对象。

这个方法只有在视图函数中出现异常了才执行，它返回的值可以是一个 `None` 也可以是一个 `HttpResponse` 对象。如果是 `HttpResponse` 对象，Django 将调用模板和中间件中的 `processresponse` 方法，并返回给浏览器，否则将默认处理异常。如果返回一个 `None`，则交给下一个中间件的 `processexception` 方法来处理异常。它的执行顺序也是按照中间件注册顺序的倒序执行。



给 MD1 和 MD2 添加上这个方法：

```
from django.utils.deprecation import MiddlewareMixin
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1 里面的 process_request")
    def process_response(self, request, response):
        print("MD1 里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)
        print("MD1 中的 process_view")
        print(view_func, view_func.__name__)
    def process_exception(self, request, exception):
        print(exception)
        print("MD1 中的 process_exception")
class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2 里面的 process_request")
        pass
    def process_response(self, request, response):
        print("MD2 里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)
        print("MD2 中的 process_view")
        print(view_func, view_func.__name__)
```

```
def process_exception(self, request, exception):
    print(exception)
    print("MD2 中的 process_exception")
```

如果视图函数中无异常，process_exception 方法不执行。

想办法，在视图函数中抛出一个异常：

```
def index(request):
    print("app01 中的 index 视图")
    raise ValueError("呵呵")
    return HttpResponse("098K")
```

在 MD1 的 process_exception 中返回一个响应对象：

```
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1 里面的 process_request")
    def process_response(self, request, response):
        print("MD1 里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)
        print("MD1 中的 process_view")
        print(view_func, view_func.__name__)

    def process_exception(self, request, exception):
        print(exception)
        print("MD1 中的 process_exception")
        return HttpResponse(str(exception)) # 返回一个响应对象
```

看输出结果：

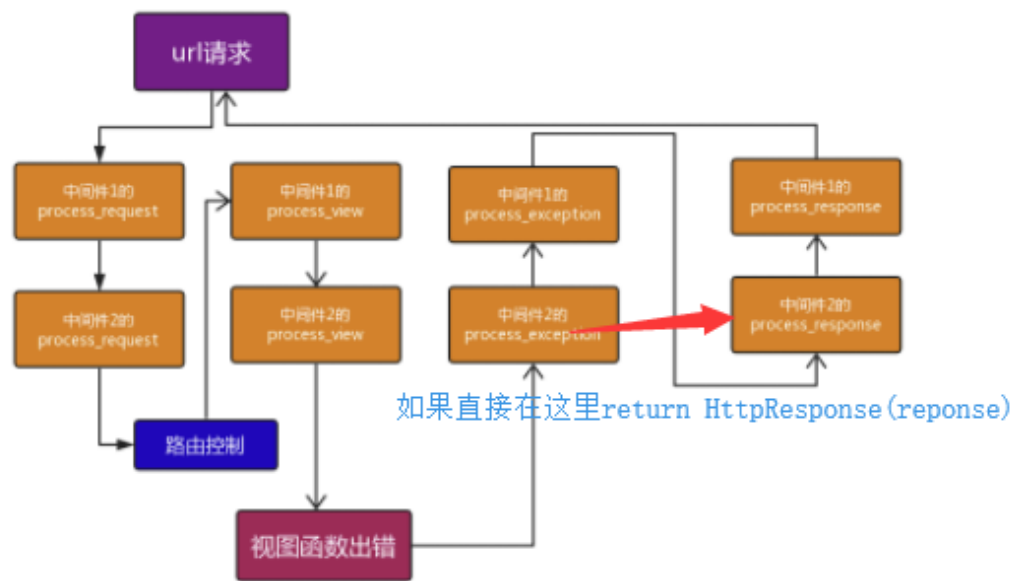
```
MD2 里面的 process_request
MD1 里面的 process_request
```

```
-----
MD2 中的 process_view
<function index at 0x0000022C09727488> index
```

```
-----
MD1 中的 process_view
<function index at 0x0000022C09727488> index
```

```
app01 中的 index 视图
呵呵
```

```
MD1 中的 process_exception
MD1 里面的 process_response
MD2 里面的 process_response
```



注意，这里并没有执行 MD2 的 `processexception` 方法，因为 MD1 中的 `processexception` 方法直接返回了一个响应对象。

processtemplateresponse（用的比较少）

`processtemplateresponse(self, request, response)`

它的参数，一个 `HttpRequest` 对象，`response` 是 `TemplateResponse` 对象（由视图函数或者中间件产生）。

`processtemplateresponse` 是在视图函数执行完成后立即执行，但是它有一个前提条件，那就是视图函数返回的对象有一个 `render()` 方法（或者表明该对象是一个 `TemplateResponse` 对象或等价方法）。

```
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1 里面的 process_request")
    def process_response(self, request, response):
        print("MD1 里面的 process_response")
        return response
    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)
        print("MD1 中的 process_view")
        print(view_func, view_func.__name__)

    def process_exception(self, request, exception):
        print(exception)
        print("MD1 中的 process_exception")
        return HttpResponse(str(exception))

    def process_template_response(self, request, response):
        print("MD1 中的 process_template_response")
        return response
```

```

class MD2(MiddlewareMixin):
    def process_request(self, request):
        print("MD2 里面的 process_request")
        pass

    def process_response(self, request, response):
        print("MD2 里面的 process_response")
        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        print("-" * 80)
        print("MD2 中的 process_view")
        print(view_func, view_func.__name__)

    def process_exception(self, request, exception):
        print(exception)
        print("MD2 中的 process_exception")

    def process_template_response(self, request, response):
        print("MD2 中的 process_template_response")
        return response

```

views.py 中:

```

def index(request):
    print("app01 中的 index 视图")

    def render():
        print("in index/render")
        return HttpResponse("098K")
    rep = HttpResponse("OK")
    rep.render = render
    return rep

```

访问 index 视图，终端输出的结果:

```

MD2 里面的 process_request
MD1 里面的 process_request
-----
MD2 中的 process_view
<function index at 0x000001C111B97488> index
-----
MD1 中的 process_view
<function index at 0x000001C111B97488> index
app01 中的 index 视图
MD1 中的 process_template_response
MD2 中的 process_template_response
in index/render
MD1 里面的 process_response
MD2 里面的 process_response

```

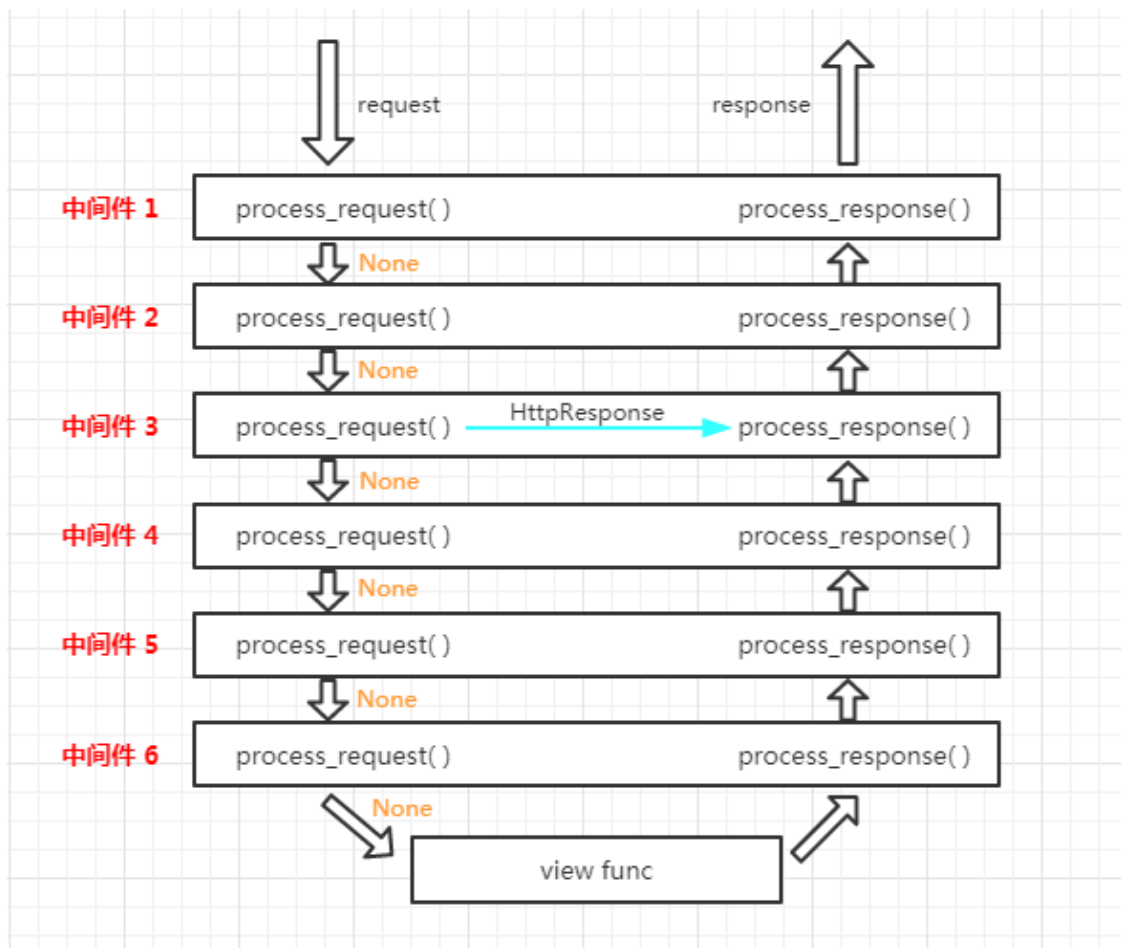
从结果看出:

视图函数执行完之后，立即执行了中间件的 `process_template_response` 方法，顺序是倒序，先执行 MD1 的，在执行 MD2 的，接着执行了视图函数返回的 `HttpResponse` 对象的 `render` 方法，返回了一个新的 `HttpResponse` 对象，接着执行中间件的 `process_response` 方法。

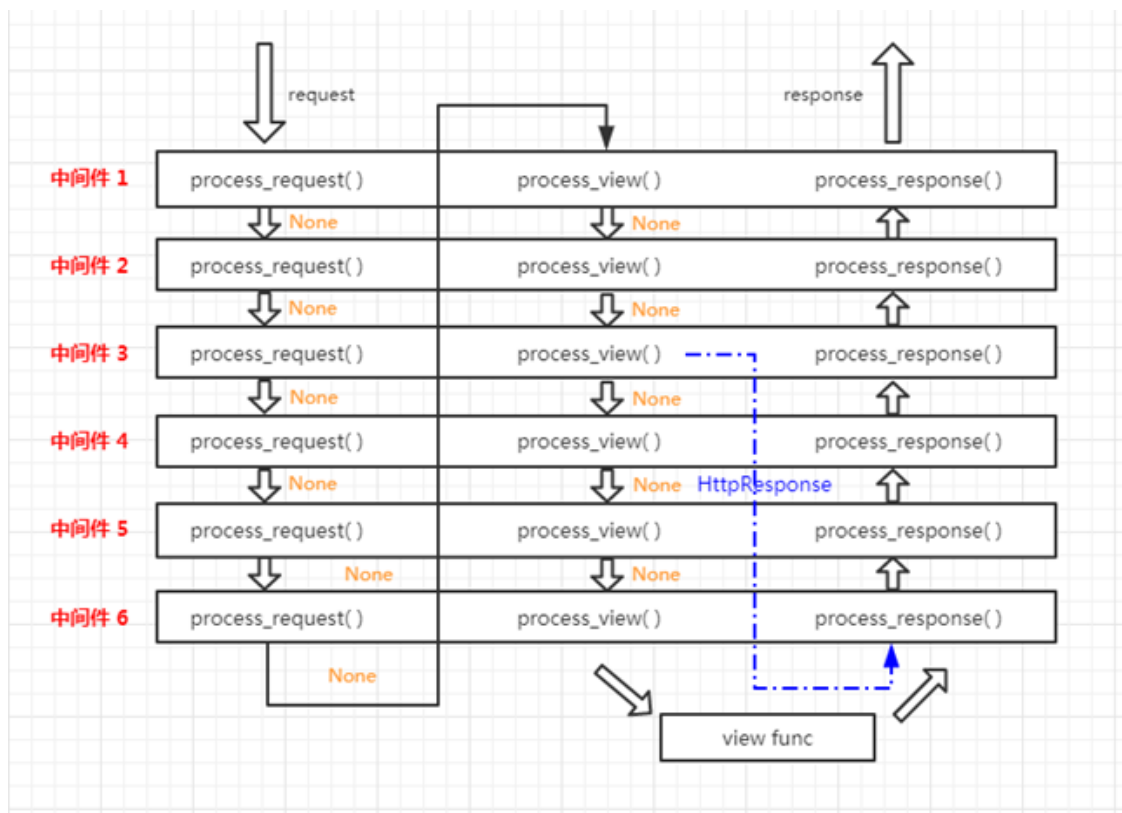
四 中间件执行流程

上一部分，我们了解了中间件中的 5 个方法，它们的参数、返回值以及什么时候执行，现在总结一下中间件的执行流程。

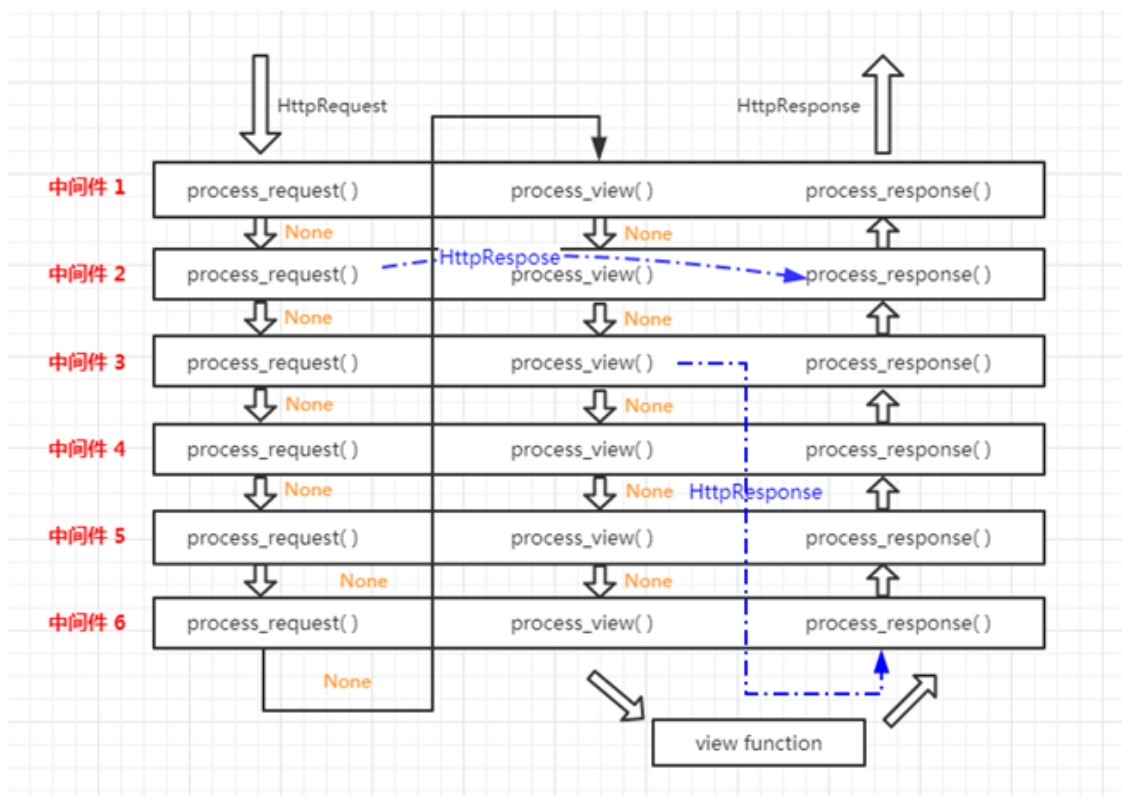
请求到达中间件之后，先按照正序执行每个注册中间件的 `process_request` 方法，`process_request` 方法返回的值是 `None`，就依次执行，如果返回的值是 `HttpResponse` 对象，不再执行后面的 `process_request` 方法，而是执行当前对应中间件的 `process_response` 方法，将 `HttpResponse` 对象返回给浏览器。也就是说：如果 MIDDLEWARE 中注册了 6 个中间件，执行过程中，第 3 个中间件返回了一个 `HttpResponse` 对象，那么第 4,5,6 中间件的 `process_request` 和 `process_response` 方法都不执行，顺序执行 3,2,1 中间件的 `process_response` 方法。

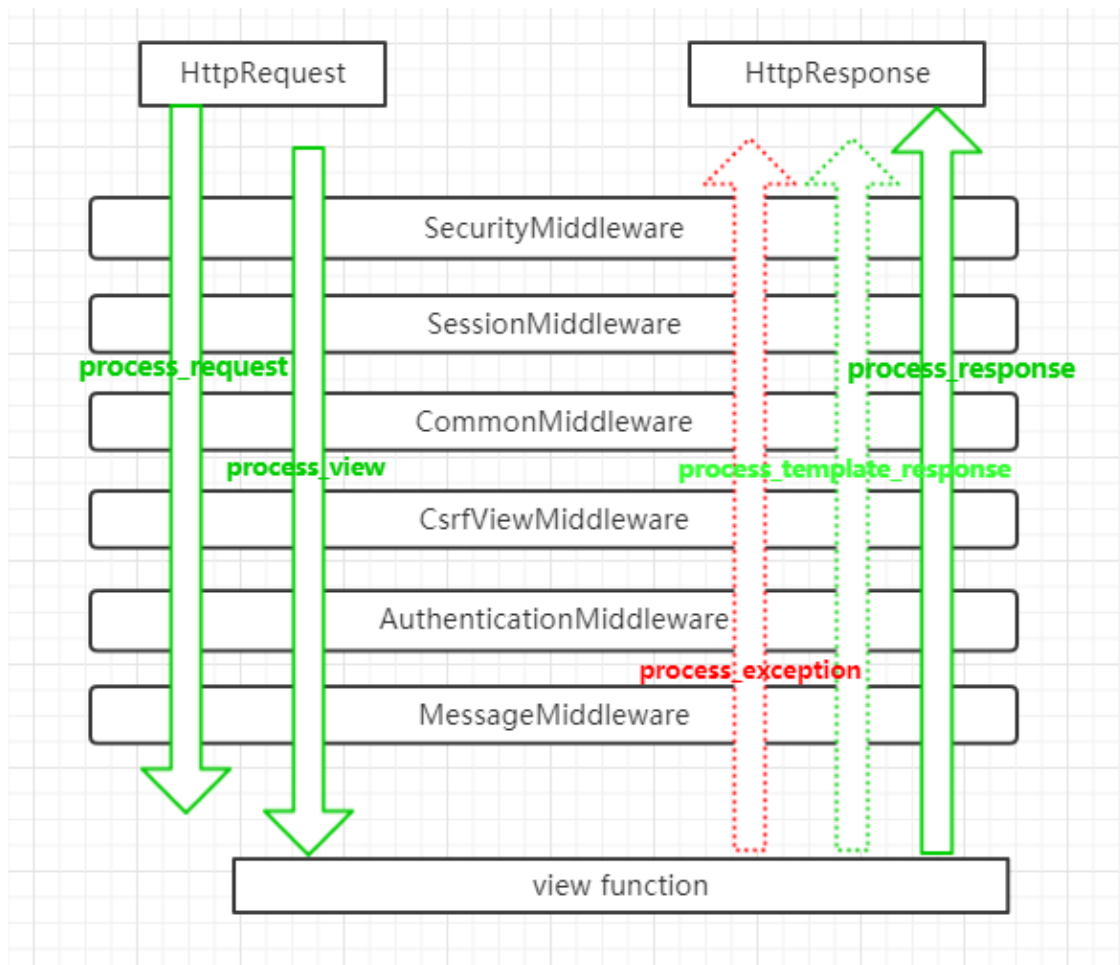


`process_request` 方法都执行完后，匹配路由，找到要执行的视图函数，先不执行视图函数，先执行中间件中的 `process_view` 方法，`process_view` 方法返回 `None`，继续按顺序执行，所有 `process_view` 方法执行完后执行视图函数。加入中间件 3 的 `process_view` 方法返回了 `HttpResponse` 对象，则 4,5,6 的 `process_view` 以及视图函数都不执行，直接从最后一个中间件，也就是中间件 6 的 `process_response` 方法开始倒序执行。



`process_exception` 和 `process_exception` 两个方法的触发是有条件的，执行顺序也是倒序。总结所有的执行流程如下：





五 中间件版登陆认证

中间件版的登录验证需要依靠 `session`，所以数据库中要有 `django_session` 表。

`urls.py`

```
from django.conf.urls import url
from app01 import views

urlpatterns = [
    url(r'^index/$', views.index),
    url(r'^login/$', views.login, name='login'),
]
```

`views.py`

```
from django.shortcuts import render, HttpResponseRedirect, redirect

def index(request):
    return HttpResponseRedirect('this is index')

def home(request):
    return HttpResponseRedirect('this is home')

def login(request):
    if request.method == "POST":
        user = request.POST.get("user")
```

```

pwd = request.POST.get("pwd")

if user == "Q1mi" and pwd == "123456":
    # 设置 session
    request.session["user"] = user
    # 获取跳到登陆页面之前的 URL
    next_url = request.GET.get("next")
    # 如果有，就跳转回登陆之前的 URL
    if next_url:
        return redirect(next_url)
    # 否则默认跳转到 index 页面
    else:
        return redirect("/index/")
return render(request, "login.html")

```

login.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>登录页面</title>
</head>
<body>
<form action="{% url 'login' %}">
    <p>
        <label for="user">用户名: </label>
        <input type="text" name="user" id="user">
    </p>
    <p>
        <label for="pwd">密 码: </label>
        <input type="text" name="pwd" id="pwd">
    </p>
    <input type="submit" value="登录">
</form>
</body>
</html>

```

middlewares.py

```

class AuthMD(MiddlewareMixin):
    white_list = ['/login/', ] # 白名单
    balck_list = ['/black/', ] # 黑名单
    def process_request(self, request):
        from django.shortcuts import redirect, HttpResponseRedirect

        next_url = request.path_info
        print(request.path_info, request.get_full_path())

        if next_url in self.white_list or request.session.get("user"):
            return
        elif next_url in self.balck_list:
            return HttpResponseRedirect('This is an illegal URL')

```

```
else:
    return redirect("/login/?next={}".format(next_url))
```

在 settings.py 中注册

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'middlewares.AuthMD',
]
```

AuthMD 中间件注册后，所有的请求都要走 AuthMD 的 process_request 方法。

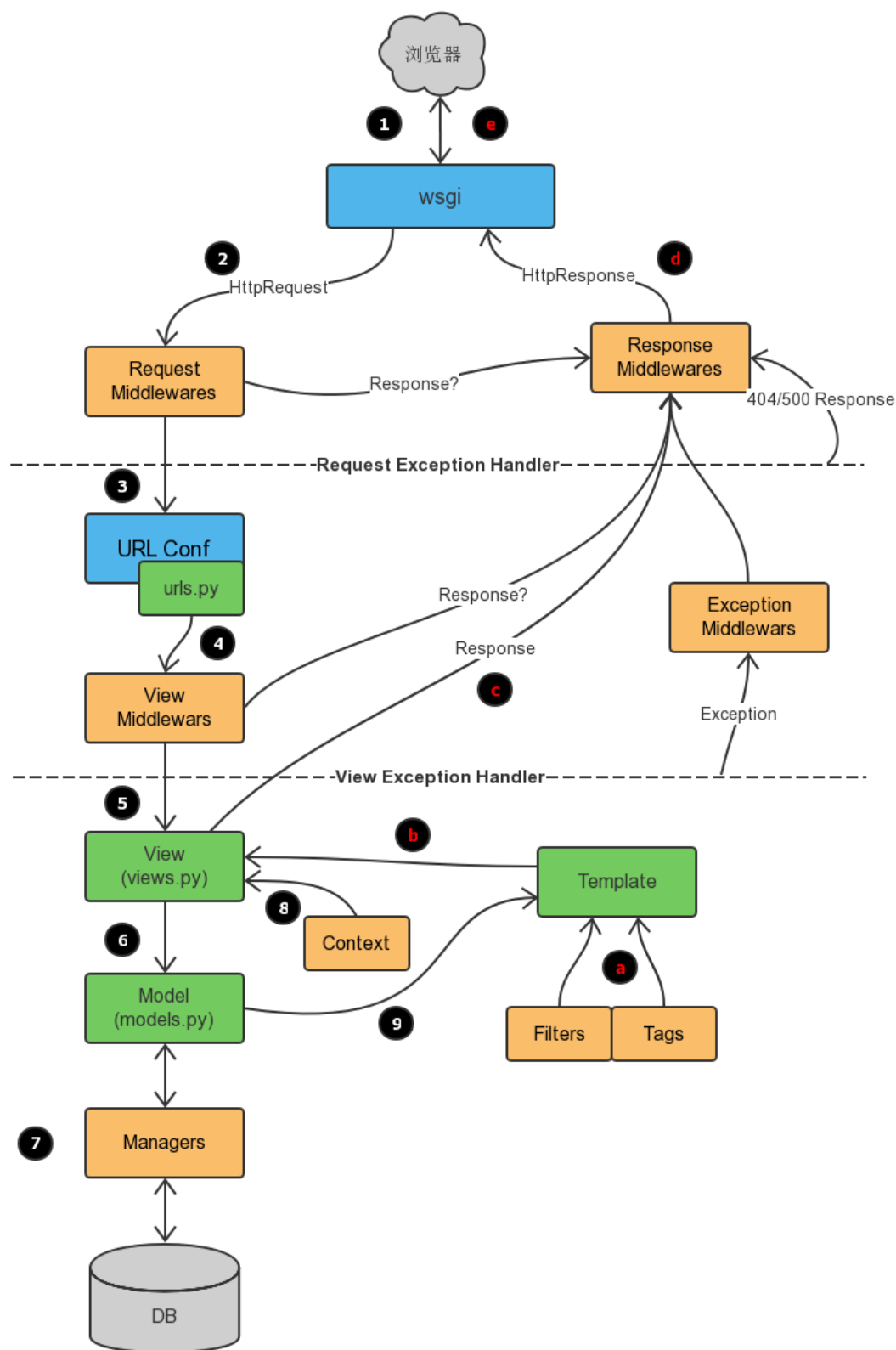
访问的 URL 在白名单内或者 session 中有 user 用户名，则不做阻拦走正常流程；

如果 URL 在黑名单中，则返回 This is an illegal URL 的字符串；

正常的 URL 但是需要登录后访问，让浏览器跳转到登录页面。

注：AuthMD 中间件中需要 session，所以 AuthMD 注册的位置要在 session 中间的下方。

附：Django 请求流程图



应用案例

1、做 IP 访问频率限制

某些 IP 访问服务器的频率过高，进行拦截，比如限制每分钟不能超过 20 次。

2、URL 访问过滤

如果用户访问的是 login 视图（放过）

如果访问其他视图，需要检测是不是有 session 认证，已经有了放行，没有返回 login，这样就省得在多个视图函数上写装饰器了！

源码试读

作为延伸扩展内容，有余力的同学可以尝试着读一下以下两个自带的中间件：

```
'django.contrib.sessions.middleware.SessionMiddleware',  
'django.contrib.auth.middleware.AuthenticationMiddleware',
```

七 xxx

八 xxx

中间件版的登录验证需要依靠 session，所以数据库中要有 django_session 表。

urls.py

```
from django.conf.urls import url  
from app01 import views  
  
urlpatterns = [  
    url(r'^index/$', views.index),  
    url(r'^login/$', views.login, name='login'),  
]
```

views.py

```
from django.shortcuts import render, HttpResponseRedirect, redirect  
def index(request):  
    return HttpResponseRedirect('this is index')  
def home(request):  
    return HttpResponseRedirect('this is home')  
def login(request):  
    if request.method == "POST":  
        user = request.POST.get("user")  
        pwd = request.POST.get("pwd")  
        if user == "Q1mi" and pwd == "123456":  
            # 设置 session  
            request.session["user"] = user  
            # 获取跳到登陆页面之前的 URL  
            next_url = request.GET.get("next")  
            # 如果有，就跳转回登陆之前的 URL  
            if next_url:  
                return redirect(next_url)  
            # 否则默认跳转到 index 页面
```

```

        else:
            return redirect("/index/")
    return render(request, "login.html")

```

login.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>登录页面</title>
</head>
<body>
<form action="{% url 'login' %}">
    <p>
        <label for="user">用户名: </label>
        <input type="text" name="user" id="user">
    </p>
    <p>
        <label for="pwd">密 码: </label>
        <input type="text" name="pwd" id="pwd">
    </p>
    <input type="submit" value="登录">
</form>
</body>
</html>

```

middlewares.py

```

class AuthMD(MiddlewareMixin):
    white_list = ['/login/', ] # 白名单
    balck_list = ['/black/', ] # 黑名单

    def process_request(self, request):
        from django.shortcuts import redirect, HttpResponseRedirect

        next_url = request.path_info
        print(request.path_info, request.get_full_path())

        if next_url in self.white_list or request.session.get("user"):
            return
        elif next_url in self.balck_list:
            return HttpResponseRedirect('This is an illegal URL')
        else:
            return redirect("/login/?next={}".format(next_url))

```

在 settings.py 中注册

```

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',

```

```
    'django.contrib.messages.middleware.MessageMiddleware',  
    'middlewares.AuthMD',  
]
```

AuthMD 中间件注册后，所有的请求都要走 AuthMD 的 `process_request` 方法。

访问的 URL 在白名单内或者 session 中有 user 用户名，则不做阻拦走正常流程；

如果 URL 在黑名单中，则返回 `This is an illegal URL` 的字符串；

正常的 URL 但是需要登录后访问，让浏览器跳转到登录页面。

注：AuthMD 中间件中需要 session，所以 AuthMD 注册的位置要在 session 中间的下方。

3. REST 数据接口如何设计（URL、域名、版本、过滤、状态码、安全性）：

建议阅读阮一峰的《[RESTful API 设计指南](#)》。

REST 接口规范总结

1 REST

REST 是一种软件架构风格，如果你的接口是 REST 接口，那么该接口可被认为是 REST 风格的。REST 接口是围绕资源展开的，HTTP 的 URL 即资源，利用 HTTP 的协议，其实 rest 本也可以和 HTTP 无关，但是现在大家普遍的使用 REST 都是依托于 HTTP 协议。

2 URI 语法

URI = scheme “://” authority “/” path “?” query

scheme: 指底层用的协议，如 http、https、ftp

host: 服务器的 IP 地址或者域名

port: 端口，http 中默认 80

path: 访问资源的路径，就是咱们各种 web 框架中定义的 route 路由

query: 为发送给服务器的参数

fragment: 锚点，定位到页面的资源，锚点为资源 id

3 URL 路径

3.1 endpoint

路径又称“终点”（endpoint），表示 API 的具体网址。

在 RESTful 架构中，每个网址代表一种资源，所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。

一般来说，数据库中的表都是同种记录的“集合”（collection），所以 API 中的名词也应该使用复数。

举例来说，有一个 API 提供学校的信息，包括各个班级，老师，同学的信息，则它的路径应该设计成下面这样：

```
https://school/v1/classes  
https://school/v1/teachers  
https://school/v1/students
```

3.1 URL 规则

1 名词对应数据库中的表

2 URL 中不能有动词

3 URL 结尾不应该包含斜杠“/”

4 正斜杠分隔符“/”必须用来指示层级关系

5 使用连字符“-”来提高 URL 的可读性，而不是使用下划线“_”

6 URL 路径中首选小写字母

7 URL 路径名词均为复数

4 HTTP 动词

对于 URL 资源的具体操作类型，由 HTTP 动词表示。

常用的 HTTP 动词有下面五个（括号里是对应的 SQL 命令）。

GET（SELECT）：从服务器取出资源（一项或多项）。

POST（CREATE）：在服务器新建一个资源。

PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。

PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）。

DELETE（DELETE）：从服务器删除资源。

5 资源过滤

如果只需要查询某些条件的数据 API 应该提供参数，过滤返回结果。

下面是一些常见的参数：

?limit=10：指定返回记录的数量

?offset=10：指定返回记录的开始位置。

?page=2&per_page=100：指定第几页，以及每页的记录数。

?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。

?teacher_subject=语文：指定筛选条件

6 返回

返回状态码推荐标准 HTTP 状态码

有很多服务器将返回状态码一直设为 200，然后在返回 body 里面自定义一些状态码来表示服务器返回结果的状态码。

由于 REST API 是直接使用的 HTTP 协议，所以它的状态码也要尽量使用 HTTP 协议的状态码。

200 OK 服务器返回用户请求的数据，该操作是幂等的

201 CREATED 新建或者修改数据成功

204 NOT CONTENT 删除数据成功

400 BAD REQUEST 用户发出的请求有问题，该操作是幂等的

401 Unauthorized 表示用户没有认证，无法进行操作

403 Forbidden 用户访问是被禁止的

422 Unprocesable Entity 当创建一个对象时，发生一个验证错误

500 INTERNAL SERVER ERROR 服务器内部错误，用户将无法判断发出的请求是否成功

503 Service Unavailable 服务不可用状态，多半是因为服务器问题，例如 CPU 占用率大，等等

.....

返回结果

GET /teachers 返回资源列表

GET /teachers/:id 返回单独的资源

POST /teachers 返回新生成的资源对象

PUT /teachers/:id 返回完整的资源对象

PATCH /teachers/:id 返回被修改的属性

DELETE /teachers/:id 返回一个空文档一、协议

RESTful API:与用户的通信协议，总是使用 [HTTPs 协议](#)。

二、域名

应该尽量将 API 部署在专用域名之下。

`https://api.example.com`

如果确定 API 很简单，不会有进一步扩展，可以考虑放在主域名下。

`https://example.org/api/`

三、版本（Versioning）

应该将 API 的版本号放入 URL。

`https://api.example.com/v1/`

另一种做法是，将版本号放在 HTTP 头信息中，但不如放入 URL 方便和直观。Github 采用这种做法。

四、路径（Endpoint）

路径又称“终点”（endpoint），表示 API 的具体网址。

在 RESTful 架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的“集合”（collection），所以 API 中的名词也应该使用复数。

举例来说，有一个 API 提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

- `https://api.example.com/v1/zoos`
- `https://api.example.com/v1/animals`
- `https://api.example.com/v1/employees`

五、HTTP 动词

对于资源的具体操作类型，由 HTTP 动词表示。

常用的 HTTP 动词有下面五个（括号里是对应的 SQL 命令）。

- GET（SELECT）：从服务器取出资源（一项或多项）。
- POST（CREATE）：在服务器新建一个资源。
- PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。
- PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）。
- DELETE（DELETE）：从服务器删除资源。

还有两个不常用的 HTTP 动词。

- HEAD：获取资源的元数据。
- OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

下面是一些例子。

- GET /zoos：列出所有动物园
- POST /zoos：新建一个动物园
- GET /zoos/ID：获取某个指定动物园的信息
- PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）
- PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）
- DELETE /zoos/ID：删除某个动物园

- GET /zoos/ID/animals: 列出某个指定动物园的所有动物
- DELETE /zoos/ID/animals/ID: 删除某个指定动物园的指定动物

六、过滤信息 (Filtering)

如果记录数量很多，服务器不可能都将它们返回给用户。API 应该提供参数，过滤返回结果。

下面是一些常见的参数。

- ?limit=10: 指定返回记录的数量
- ?offset=10: 指定返回记录的开始位置。
- ?page=2&per_page=100: 指定第几页，以及每页的记录数。
- ?sortby=name&order=asc: 指定返回结果按照哪个属性排序，以及排序顺序。
- ?animaltypeid=1: 指定筛选条件

参数的设计允许存在冗余，即允许 API 路径和 URL 参数偶尔有重复。比如，GET /zoo/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。

七、状态码 (Status Codes)

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的 HTTP 动词）。

- 200 OK - [GET]: 服务器成功返回用户请求的数据，该操作是幂等的 (Idempotent)。
- 201 CREATED - [POST/PUT/PATCH]: 用户新建或修改数据成功。
- 202 Accepted - [*]: 表示一个请求已经进入后台排队（异步任务）
- 204 NO CONTENT - [DELETE]: 用户删除数据成功。
- 400 INVALID REQUEST - [POST/PUT/PATCH]: 用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
- 401 Unauthorized - [*]: 表示用户没有权限（令牌、用户名、密码错误）。
- 403 Forbidden - [*] 表示用户得到授权（与 401 错误相对），但是访问是被禁止的。
- 404 NOT FOUND - [*]: 用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
- 406 Not Acceptable - [GET]: 用户请求的格式不可得（比如用户请求 JSON 格式，但是只有 XML 格式）。
- 410 Gone -[GET]: 用户请求的资源被永久删除，且不会再得到的。
- 422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
- 500 INTERNAL SERVER ERROR - [*]: 服务器发生错误，用户将无法判断发出的请求是否成功。

状态码的完全列表参见[这里](#)。

八、错误处理 (Error handling)

如果状态码是 4xx，就应该向用户返回出错信息。一般来说，返回的信息中将 error 作为键名，出错信息作为键值即可。

```
{  
  error: "Invalid API key"  
}
```

九、返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

- GET /collection: 返回资源对象的列表（数组）
- GET /collection/resource: 返回单个资源对象

- POST /collection: 返回新生成的资源对象
- PUT /collection/resource: 返回完整的资源对象
- PATCH /collection/resource: 返回完整的资源对象
- DELETE /collection/resource: 返回一个空文档

十、Hypermedia API

RESTful API 最好做到 Hypermedia，即返回结果中提供链接，连向其他 API 方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向 api.example.com 的根目录发出请求，会得到这样一个文档。

```
{
  "link": {
    "rel": "collection https://www.example.com/zoos",
    "href": "https://api.example.com/zoos",
    "title": "List of zoos",
    "type": "application/vnd.yourformat+json"
  }
}
```

上面代码表示，文档中有一个 link 属性，用户读取这个属性就知道下一步该调用什么 API 了。rel 表示这个 API 与当前网址的关系（collection 关系，并给出该 collection 的网址），href 表示 API 的路径，title 表示 API 的标题，type 表示返回类型。

Hypermedia API 的设计被称为 HATEOAS。Github 的 API 就是这种设计，访问 api.github.com 会得到一个所有可用 API 的网址列表。

```
{
  "current_user_url": "https://api.github.com/user",
  "authorizations_url": "https://api.github.com/authorizations",
  // ...
}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问 api.github.com/user，然后就得到了下面结果。

```
{
  "message": "Requires authentication",
  "documentation_url": "https://developer.github.com/v3"
}
```

上面代码表示，服务器给出了提示信息，以及文档的网址。

十一、其他

- (1) API 的身份认证应该使用 OAuth 2.0 框架。
- (2) 服务器返回的数据格式，应该尽量使用 JSON，避免使用 XML。

URI 格式规范

- URI(Uniform Resource Identifiers) 统一资源标示符
- URL(Uniform Resource Locator) 统一资源定位符

URI 的格式定义如下：

URI = scheme "://" authority "/" path ["?" query] ["#" fragment]

URL 是 URI 的一个子集(一种具体实现), 对于 REST API 来说一个资源一般对应一个唯一的 URI(URL)。在 URI 的设计中, 我们会遵循一些规则, 使接口看起透明易读, 方便使用者调用。

- 关于分隔符"/"的使用

"/"分隔符一般用来对资源层级的划分, 例如 `http://api.canvas.restapi.org/shapes/polygons/quadrilaterals/squares`

对于 REST API 来说, "/"只是一个分隔符, 并无其他含义。为了避免混淆, "/"不应该出现在 URL 的末尾。

例如以下两个地址实际表示的都是同一个资源:

`http://api.canvas.restapi.org/shapes/`

`http://api.canvas.restapi.org/shapes`

REST API 对 URI 资源的定义具有唯一性, 一个资源对应一个唯一的地址。为了使接口保持清晰干净, 如果访问到末尾包含 "/" 的地址, 服务端应该 301 到没有 "/" 的地址上。当然这个规则也仅限于 REST API 接口的访问, 对于传统的 WEB 页面服务来说, 并不一定适用这个规则。

1234567

- URI 中尽量使用连字符"-"代替下划线"_"的使用

连字符"-"一般用来分割 URI 中出现的字符串(单词), 来提高 URI 的可读性, 例如:

`http://api.example.restapi.org/blogs/mark-masse/entries/this-is-my-first-post`

使用下划线"_"来分割字符串(单词)可能会和链接的样式冲突重叠, 而影响阅读性。但实际上, "-"和"_"对 URL 中字符串的分割语意上还是有些差异的: "-"分割的字符串(单词)一般各自都具有独立的含义, 可参见上面的例子。而"_"一般用于对一个整体含义的字符串做了层级的分割, 方便阅读, 例如你想在 URL 中体现一个 ip 地址的信息: `210_110_25_88` .

1234

- URI 中统一使用小写字母

根据 RFC3986 定义, URI 是对大小写敏感的, 所以为了避免歧义, 我们尽量用小写字符。但主机名(Host)和 scheme (协议名称:`http/ftp/...`) 对大小写是不敏感的。

1

- URI 中不要包含文件(脚本)的扩展名

例如 `.php .json` 之内的就不要出现了, 对于接口来说没有任何实际的意义。如果是想对返回的数据内容格式标示的话, 通过 HTTP Header 中的 Content-Type 字段更好一些。

1

资源的原型

- 文档(Document)

文档是资源的单一表现形式, 可以理解为一个对象, 或者数据库中的一条记录。在请求文档时, 要么返回文档对应的数据, 要么会返回一个指向另外一个资源(文档)的链接。以下是几个基于文档定义的 URI 例子:

`http://api.soccer.restapi.org/leagues/seattle` `http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet` `http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players/mike`

12

- 集合(Collection)

集合可以理解为是资源的一个容器(目录), 我们可以向里面添加资源(文档)。例如:

`http://api.soccer.restapi.org/leagues` `http://api.soccer.restapi.org/leagues/seattle/tea`

```
ms http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players
12
```

- 仓库(Store)

仓库是客户端来管理的一个资源库，客户端可以向仓库中新增资源或者删除资源。客户端也可以批量获取到某个仓库下的所有资源。仓库中的资源对外的访问不会提供单独 URI 的，客户端在创建资源时候的 URI 除外。

例如：

```
PUT /users/1234/favorites/alonso
```

上面的例子我们可以理解为，我们向一个 id 是 1234 的用户的仓库(收藏夹)中，添加了一个名为 alonso 的资源。通俗点儿说：就是用户收藏了一个自己喜爱的球员阿隆索。

```
123
```

- 控制器(Controller)

控制器资源模型，可以执行一个方法，支持参数输入，结果返回。是为了除了标准操作：增删改查(CRUD)以外的一些逻辑操作。控制器(方法)一般定义于 URI 中末尾，并且不会有子资源(控制器)。例如我们向用户重发 ID 为 245743 的消息：

```
POST /alerts/245743/resend
```

```
12
```

URI 命名规范

- 文档(Document)类型的资源用**名词(短语)单数**命名
- 集合(Collection)类型的资源用**名词(短语)复数**命名
- 仓库(Store)类型的资源用**名词(短语)复数**命名
- 控制器(Controller)类型的资源用**动词(短语)**命名
- URI 中有些字段可以是变量，在实际使用中可以按需替换

例如一个资源 URI 可以这样定义：

```
http://api.soccer.restapi.org/leagues/{leagueId}/teams/{teamId}/players/{playerId}
```

其中：leagueId,teamId,playerId 是变量(数字，字符串都类型都可以)。

```
123
```

- **CRUD** 的操作不要体现在 URI 中，HTTP 协议中的操作符已经对 CRUD 做了映射。

CRUD 是创建，读取，更新，删除这四个经典操作的简称

例如删除的操作用 REST 规范执行的话，应该是这个样子：

```
DELETE /users/1234
```

以下是几个错误的示例：

```
GET /deleteUser?id=1234
```

```
GET /deleteUser/1234
```

```
DELETE /deleteUser/1234
```

```
POST /users/1234/delete
```

```
123456789
```

URI 的 query 字段

```
http://api.college.restapi.org/students/morgan/send-sms http://api.college.restapi.org/students/morgan/send-sms?text=hello
```

```
1
```

以上的两个 URI 看起来很像，但实际的含义是有差别的。第一个 URI 是一个发送消息的 Controller 类型的 API，第二个 URI 是发送一个 text 的内容是 hello 的消息。

在 REST 中,query 字段一般作为查询的参数补充，也可以帮助标示一个唯一的资源。但需要注意的是，作为一个提供查询功能的 URI，无论是否有 query 条件，我们都应该保证结果的唯一性，一个 URI 对应的返回数据是不应该被改变的(在资源没有修改的情况下)。HTTP 中的缓存也可能缓存查询结果，这个也是我们需要知道的。

- Query 参数可以作为 Collection 或 Store 类型资源的过滤条件来使用
例如：

GET /users //返回所有用户列表

GET /users?role=admin //返回权限为 admin 的用户列表

12

- Query 参数可以作为 Collection 或 Store 资源列表分页标示使用

如果是一个简单的列表操作，可以这样设计：

GET /users?pageSize=25&pageStartIndex=50

如果是一个复杂的列表或查询操作的话，我们可以为资源设计一个 Collection，因为复杂查询可能会涉及较多的参数，建议使用 Post 的方式传入，例如这样：

POST /users/search

1234

HTTP 交互设计

HTTP 请求方法的使用

- **GET** 方法用来获取资源
- **PUT** 方法可用来新增/更新 Store 类型的资源
- **PUT** 方法可用来更新一个资源
- **POST** 方法可用来创建一个资源
- **POST** 方法可用来触发执行一个 Controller 类型资源
- **DELETE** 方法用于删除资源

一旦资源被删除，GET/HEAD 方法访问被删除的资源时，要返回 404

DELETE 是一个比较纯粹的方法，我们不能对其做任何的重构或者定义，不可附加其它状态条件，如果我们希望"软"删除一个资源，则这种需求应该由 Controller 类资源来实现。

12

HTTP 响应状态码的使用

- **200 ("OK")** 用于一般性的成功返回
- **200 ("OK")** 不可用于请求错误返回
- **201 ("Created")** 资源被创建
- **202 ("Accepted")** 用于 Controller 控制类资源异步处理的返回，仅表示请求已经收到。对于耗时比较久的处理，一般用异步处理来完成

- **204 (“No Content”)** 此状态可能会出现在 PUT、POST、DELETE 的请求中，一般表示资源存在，但消息体中不会返回任何资源相关的状态或信息。
- **301 (“Moved Permanently”)** 资源的 URI 被转移，需要使用新的 URI 访问
- **302 (“Found”)** 不推荐使用，此代码在 HTTP1.1 协议中被 303/307 替代。我们目前对 302 的使用和最初 HTTP1.0 定义的语意是有出入的，应该只有在 GET/HEAD 方法下，客户端才能根据 Location 执行自动跳转，而我们目前的客户端基本上是不会判断原请求方法的，无条件的执行临时重定向
- **303 (“See Other”)** 返回一个资源地址 URI 的引用，但不强制要求客户端获取该地址的状态(访问该地址)
- **304 (“Not Modified”)** 有一些类似于 204 状态，服务器端的资源与客户端最近访问的资源版本一致，并无修改，不返回资源消息体。可以用来降低服务端的压力
- **307 (“Temporary Redirect”)** 目前 URI 不能提供当前请求的服务，临时性重定向到另外一个 URI。在 HTTP1.1 中 307 是用来替代早期 HTTP1.0 中使用不当的 302
- **400 (“Bad Request”)** 用于客户端一般性错误返回,在其它 4xx 错误以外的错误，也可以使用 400，具体错误信息可以放在 body 中
- **401 (“Unauthorized”)** 在访问一个需要验证的资源时，验证错误
- **403 (“Forbidden”)** 一般用于非验证性资源访问被禁止，例如对于某些客户端只开放部分 API 的访问权限，而另外一些 API 可能无法访问时，可以给予 403 状态
- **404 (“Not Found”)** 找不到 URI 对应的资源
- **405 (“Method Not Allowed”)** HTTP 的方法不支持，例如某些只读资源，可能不支持 POST/DELETE。但 405 的响应 header 中必须声明该 URI 所支持的方法
- **406 (“Not Acceptable”)** 客户端所请求的资源数据格式类型不被支持，例如客户端请求数据格式为 application/xml，但服务器端只支持 application/json
- **409 (“Conflict”)** 资源状态冲突，例如客户端尝试删除一个非空的 Store 资源
- **412 (“Precondition Failed”)** 用于有条件的操作不被满足时
- **415 (“Unsupported Media Type”)** 客户所支持的数据类型，服务端无法满足
- **500 (“Internal Server Error”)** 服务器端的接口错误，此错误于客户端无关

原数据设计

HTTP Headers

- **Content-Type** 标示 body 的数据格式
- **Content-Length** body 数据体的大小，客户端可以根据此标示检验读取到的数据是否完整，也可以通过 Header 判断是否需要下载可能较大的数据体
- **Last-Modified** 用于服务器端的响应，是一个资源最后被修改的时间戳，客户端(缓存)可以根据此信息判断是否需要重新获取该资源

- **ETag** 服务器端资源版本的标示，客户端(缓存)可以根据此信息判断是否需要重新获取该资源，需要注意的是，ETag 如果通过服务器随机生成，可能会存在多个主机对同一个资源产生不同 ETag 的问题
- **Store** 类型的资源要支持有条件的 PUT 请求

假设有两个客户端 **client#1/#2** 都向一个 **Store** 资源提交 PUT 请求，服务端是无法清楚的判断是要 **insert** 还是要 **update** 的，所以我们要在 **header** 中加入条件标示 **if-Match**, **If-Unmodified-Since** 来明确是本次调用 API 的意图。例如：

client#1 第一次向服务端发起一个请求 **PUT /objects/2113** 此时 **2113** 资源还不存在，那服务端会认为本次请求是一个 **insert** 操作，完成后，会返回 **201** (“Created”)

client#2 再一次向服务端发起同一个请求 **PUT /objects/2113** 时，因 **2113** 资源已存在，服务端会返回 **409** (“Conflict”)

为了能让 **client#2** 的请求成功，或者说我们要清楚的表明本次操作是一次 **update** 操作，我们必须在 **header** 中加入一些条件标示，例如 **if-Match**。我们需要给出资源的 **ETag(if-Match: Etag)**，来表明我们希望更新资源的版本，如果服务端版本一致，会返回 **200** (“OK”) 或者 **204** (“No Content”)。如果服务端发现指定的版本与当前资源版本不一致，会返回 **412** (“Precondition Failed”)
1234567

- **Location** 在响应 **header** 中使用，一般为客户端感兴趣的资源 URI,例如在成功创建一个资源后，我们可以把新的资源 URI 放在 **Location** 中，如果是一个异步创建资源的请求，接口在响应 **202** (“Accepted”)的同时可以给予客户端一个异步状态查询的地址
- **Cache-Control, Expires, Date** 通过缓存机制提升接口响应性能,同时根据实际需要也可以禁止客户端对接口请求做缓存。对于 **REST** 接口来说，如果某些接口实时性要求不高的情况下，我们可以使用 **max-age** 来指定一个小的缓存时间，这样对客户端和服务端双方都是有利的。一般来说只对 **GET** 方法且返回 **200** 的情况下使用缓存，在某些情况下我们也可以对返回 **3xx** 或者 **4xx** 的情况下做缓存，可以防范错误访问带来的负载。
- 我们可以自定义一些头信息，作为客户端和服务端间的通信使用，但不能改变 **HTTP** 方法的性质。自定义头尽量简单明了，不要用 **body** 中的信息对其作补充说明。

数据媒体类型(Media Type)

定义如下：

Content-Type: type "/" subtype *(";" parameter)

两个实例：

Content-type: text/html; charset=ISO-8859-4

Content-type: text/plain; charset="us-ascii"

1234

type 主类型一般为：application, audio, image, message, model, multipart, text, video。REST 接口的主类型一般使用 **application**

数据媒体类型(Media Type)设计

- 设计上来说，服务器端可以支持多种媒体类型
- 可以通过 **URI** 的查询字段来指定客户端希望的数据类型

GET /bookmarks/mikemassedotcom?accept=application/xml

1

数据媒体格式的设计

body 的媒体格式

- json 是一种流行且轻便友好的格式，json 是一种无序的键值对的集合，其中 key 是需要用双引号引起来的，value 如果是数字可以不用双引号，如果是非数字的格式需要使用双引号。

这是一个 json 格式的例子：

```
{
  "firstName" : "Osvaldo",
  "lastName" : "Alonso", "firstNamePronunciation" : "ahs-VAHL-doe", "number" : 6,
  "birthDate" : "1985-11-11"
}
123456
```

- json 是允许大小写混用命名的，但要避免使用特殊符号
- 除了 json 我们也可以使用其他常用的格式，例如 xml,html 等
- body 本身只应包含资源相关的信息，不要附加其它传输状态的信息

错误响应描述

- 错误信息的格式应该保持一致，例如用以下方式(json 格式):

```
{
  "id" : Text, //错误唯一标示 id
  "description" : Text //错误具体描述
}
```

如果有多个错误，可以用 json 数组来描述：

```
{
  "elements" : [
    {
      "id" : "Update Failed",
      "description" : "Failed to update /users/1234"
    }
  ]
}
1234567891011121314
```

- 错误类型需要保持统一

客户端关注的问题

接口版本管理

- 一个资源，只用一种单一的 URI 来标示，资源的版本不应该体现在 URI 中
- 资源的版本是可以由客户端来指定的，并且提供向后兼容
- ETag 可以用来管理资源的版本，有助于客户端缓存的应用

接口的安全

- 使用 OAuth 认证，对敏感资源保护
- 使用 API 管理策略，或管理平台（Apigee, Mashery）

接口数据响应的结构

- 客户端可以指定接口返回需要的资源字段，或者指定不希望返回的字段，这样有助于提升接口交互的效率，较少带宽的浪费

只获许部分字段：

GET /students/morgan?fields=(firstName, birthDate)

12

不希望获取某些字段：

GET /students/morgan?fields=!(address,schedule!(wednesday, friday))

- 资源数据中可以包含嵌入式链接，用来描述查询资源的子集，我们也可以传入相关参数，要求服务端替换链接为实际的数据

```
{
  "firstName" : "Morgan",
  "birthDate" : "1992-07-31",
  # Other fields...
  "links" : {
    "self" : {
      "href" : "http://api.college.restapi.org/students/morgan",
      "rel" : "http://api.relations.wrml.org/common/self"
    },
    "favoriteClass" : {
      "href" : "http://api.college.restapi.org/classes/japn-301",
      "rel" : "http://api.relations.wrml.org/college/favoriteClass"
    },
    # Other links...
  }
}
```

如果我们传入 `embed=(favoriteClass)` 的参数，返回的数据中将用实际的内容替换 `links` 里的对应的潜在资源：

Request

GET /students/morgan?embed=(favoriteClass)

Response

```
{
  "firstName" : "Morgan",
  "birthDate" : "1992-07-31",
  "favoriteClass" : { //需要返回的嵌入数据
    "id" : "japn-301",
    "name" : "Third-Year Japanese",
    "links" : {
      "self" : {
        "href" : "http://api.college.restapi.org/classes/japn-301",
        "rel" : "http://api.relations.wrml.org/common/self"
      }
    }
  }
}
```

```
# Other fields...
"links" : {
  "self" : {
    "href" : "http://api.college.restapi.org/students/morgan",
    "rel" : "http://api.relations.wrml.org/common/self"
  },
  # 之前的嵌入式链接 favoriteClass, 已被替换为实体数据
  # Other links...
}
}
```

#其中嵌入式链接信息中的 `rel` ,一般是对 `href` 资源如何交互的描述,例如是通过 `GET` 还是 `POST` 方法,可以是以下的结构:

```
{
  "name": "morgan",
  "method": "GET",
  ... #其它描述字段
}
```

12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455

JavaScript 客户端

目前主流的浏览器对 JavaScript 的支持越来越完善,因此对于 WEB 应用来说,我们完全可以把客户端看成一个 JavaScript 客户端。

- 一般浏览器对于跨域的操作都有一定的安全策略,通常我们可以使用 JSONP 来解决跨域接口访问的限制
- 通过 CORS(Cross-Origin Resource Sharing)来解决跨域访问,此方法与 JSONP 相比,支持更多的方法,JSONP 只能用于 GET 请求,一般现代的浏览器会支持 CORS 的方式

一、重要概念: **REST**,即 **Representational State Transfer** 的缩写。我对这个词组的翻译是"表现层状态转化"。

Resource (资源) : 对象的单个实例。例如,一只动物。它可以是一段文本、一张图片、一首歌曲、一种服务,总之就是一个具体的实在。 你可以用一个 **URI** (统一资源定位符)指向它,每种资源对应一个特定的 **URI**。要获取这个资源,访问它的 **URI** 就可以,因此 **URI** 就成了每一个资源的地址或独一无二的识别符。

集合: 对象的集合。 例如,动物。

第三方: 使用我们接口的开发者

表现层 (Representation)

"资源"是一种信息实体,它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式,叫做它的"表现层" (Representation)。

状态转化 (State Transfer)

访问一个网站,就代表了客户端和服务器的一个互动过程。在这个过程中,势必涉及到数据和状态的变

化。互联网通信协议 HTTP 协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。客户端用到的手段，只能是 HTTP 协议。具体来说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源（也可以用于更新资源），PUT 用来更新资源，DELETE 用来删除资源。

比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式；图片可以用 JPG 格式表现，也可以用 PNG 格式表现。

URI 只代表资源的实体，不代表它的形式。严格地说，有些网址最后的".html"后缀名是不必要的，因为这个后缀名表示格式，属于"表现层"范畴，而 URI 应该只代表"资源"的位置。它的具体表现形式，应该在 HTTP 请求的头信息中用 Accept 和 Content-Type 字段指定，这两个字段才是对"表现层"的描述。

综合上面的解释，我们总结一下什么是 RESTful 架构：

- （1）每一个 URI 代表一种资源；
- （2）客户端和服务端之间，传递这种资源的某种表现层；
- （3）客户端通过四个 HTTP 动词，对服务器端资源进行操作，实现"表现层状态转化"。

二、REST 接口规范

1、动作

GET（SELECT）：从服务器检索特定资源，或资源列表。

POST（CREATE）：在服务器上创建一个新的资源。

PUT（UPDATE）：更新服务器上的资源，提供整个资源。

PATCH（UPDATE）：更新服务器上的资源，仅提供更改的属性。

DELETE（DELETE）：从服务器删除资源。

首先是四个半种动作：

post、delete、put/patch、get

因为 put/patch 只能算作一类，所以将 patch 归为半个。

另外还有两个较少知名的 HTTP 动词：

HEAD - 检索有关资源的元数据，例如数据的哈希或上次更新时间。

OPTIONS - 检索关于客户端被允许对资源做什么的信息。

2、路径（接口命名）

路径又称"终点"（endpoint），表示 API 的具体网址。

在 RESTful 架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以 API 中的名词也应该使用复数。

举例来说，有一个 API 提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

接口尽量使用名词，禁止使用动词，下面是一些例子。

GET	/zoos: 列出所有动物园
POST	/zoos: 新建一个动物园
GET	/zoos/ID: 获取某个指定动物园的信息
PUT	/zoos/ID: 更新某个指定动物园的信息（提供该动物园的全部信息）
PATCH	/zoos/ID: 更新某个指定动物园的信息（提供该动物园的部分信息）
DELETE	/zoos/ID: 删除某个动物园
GET	/zoos/ID/animals: 列出某个指定动物园的所有动物
DELETE	/zoos/ID/animals/ID: 删除某个指定动物园的指定动物

反例:

```
/getAllCars  
/createNewCar  
/deleteAllRedCars
```

再比如, 某个 URI 是 /posts/show/1, 其中 show 是动词, 这个 URI 就设计错了, 正确的写法应该是 /posts/1, 然后用 GET 方法表示 show。

如果某些动作是 HTTP 动词表示不了的, 你就应该把动作做成一种资源。比如网上汇款, 从账户 1 向账户 2 汇款 500 元, 错误的 URI 是:

```
POST /accounts/1/transfer/500/to/2
```

正确的写法是把动词 transfer 改成名词 transaction, 资源不能是动词, 但是可以是一种服务:

```
POST /transaction HTTP/1.1  
Host: 127.0.0.1  
from=1&to=2&amount=500.00
```

理清资源的层次结构, 比如业务针对的范围是学校, 那么学校会是一级资源(/school), 老师(/school/teachers), 学生(/school/students)就是二级资源。

3、版本 (Versioning)

应该将 API 的版本号放入 URL。如:

```
https://api.example.com/v1/
```

另一种做法是, 将版本号放在 HTTP 头信息中, 但不如放入 URL 方便和直观。Github 采用这种做法。

4、过滤信息 (Filtering)

如果记录数量很多, 服务器不可能都将它们返回给用户。API 应该提供参数, 过滤返回结果。下面是一些常见的参数。

```
?limit=10: 指定返回记录的数量  
?offset=10: 指定返回记录的开始位置。  
?page_number=2&page_size=100: 指定第几页, 以及每页的记录数。  
?sortby=name&order=asc: 指定返回结果按照哪个属性排序, 以及排序顺序。  
?animal_type_id=1: 指定筛选条件
```

参数的设计允许存在冗余, 即允许 API 路径和 URL 参数偶尔有重复。比如, GET /zoo/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。

5、状态码（Status Codes）

状态码范围

1xx 信息，请求收到，继续处理。范围保留用于底层 HTTP 的东西，你很可能永远也用不到。

2xx 成功，行为被成功地接受、理解和采纳

3xx 重定向，为了完成请求，必须进一步执行的动作

4xx 客户端错误，请求包含语法错误或者请求无法实现。范围保留用于响应客户端做出的错误，例如。他们提供不良数据或要求不存在的东西。这些请求应该是幂等的，而不是更改服务器的状态。

5xx 范围的状态码是保留给服务器端错误用的。这些错误常常是从底层的函数抛出来的，甚至开发人员也通常没法处理，发送这类状态码的目的以确保客户端获得某种响应。

当收到 **5xx** 响应时，客户端不可能知道服务器的状态，所以这类状态码是要尽可能的避免。

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的 HTTP 动词）。

200 OK - [GET]: 服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。

201 CREATED - [POST/PUT/PATCH]: 用户新建或修改数据成功。

202 Accepted - [*]: 表示一个请求已经进入后台排队（异步任务）

204 NO CONTENT - [DELETE]: 用户删除数据成功。

400 INVALID REQUEST - [POST/PUT/PATCH]: 用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。

401 Unauthorized - [*]: 表示用户没有权限（令牌、用户名、密码错误）。

403 Forbidden - [*] 表示用户得到授权（与 **401** 错误相对），但是访问是被禁止的。

404 NOT FOUND - [*]: 用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。

406 Not Acceptable - [GET]: 用户请求的格式不可得（比如用户请求 JSON 格式，但是只有 XML 格式）。

410 Gone - [GET]: 用户请求的资源被永久删除，且不会再得到的。

422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。

500 INTERNAL SERVER ERROR - [*]: 服务器发生错误，用户将无法判断发出的请求是否成功。

502 网关错误

503 Service Unavailable

504 网关超时

错误处理（Error handling）

如果状态码是 **4xx**，就应该向用户返回出错信息。一般来说，返回的信息中将 **error** 作为键名，出错信息作为键值即可。

```
{
  error: "Invalid API key"
}
```

返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

GET /collection: 返回资源对象的列表（数组）

GET /collection/resource: 返回单个资源对象

POST /collection: 返回新生成的资源对象

PUT /collection/resource: 返回完整的资源对象

PATCH /collection/resource: 返回完整的资源对象

DELETE /collection/resource: 返回一个空文档

Hypermedia API

RESTful API 最好做到 Hypermedia，即返回结果中提供链接，连向其他 API 方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向 `api.example.com` 的根目录发出请求，会得到这样一个文档。

```
{"link": {
  "rel": "collection https://www.example.com/zoos",
  "href": "https://api.example.com/zoos",
  "title": "List of zoos",
  "type": "application/vnd.yourformat+json"
}}
```

上面代码表示，文档中有一个 `link` 属性，用户读取这个属性就知道下一步该调用什么 API 了。`rel` 表示这个 API 与当前网址的关系（`collection` 关系，并给出该 `collection` 的网址），`href` 表示 API 的路径，`title` 表示 API 的标题，`type` 表示返回类型。

Hypermedia API 的设计被称为 **HATEOAS**。Github 的 API 就是这种设计，访问 api.github.com 会得到一个所有可用 API 的网址列表。

```
{
  "current_user_url": "https://api.github.com/user",
  "authorizations_url": "https://api.github.com/authorizations",
  // ...
}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问 api.github.com/user，然后就得到了下面结果。

```
{
  "message": "Requires authentication",
  "documentation_url": "https://developer.github.com/v3"
}
```

上面代码表示，服务器给出了提示信息，以及文档的网址。

4. 使用 ORM 框架实现 CRUD 操作的相关问题。

- 如何实现多条件组合查询 / 如何执行原生的 SQL / 如何避免 N+1 查询问题：

SQL 多条件组合查询（模糊查询）:前提设定：

假如现在有一个提交表单，里面是 N 个查询的条件，用户可以只填写其中的几个条件来进行查询。（也可以不填写条件）

解决思想：

判断用户填入的条件参数不为 `null` 以及除去空格不为空，满足该条件后，使用 `sql` 语句拼凑。

解决：

首先给出 `sql` 语句前半句：

```
StringBuilder sql = new StringBuilder("select * from t_customer where 1=1");
```

后面的 `where 1=1` 是一个始终成立的条件，是为了防止用户一个条件也没有填，那么就是查询所有，即使只有 `sql` 语句的前半段，也不会出现问题！

再给出 sql 语句的后半段（后半段的存在就说明了用户填写了 1~N 个条件）
我们只需将这些条件拼凑起来即可！

使用原生的 SQL 语言方法介绍:虽然系统已经将数据库操作进行了 OOP 封装，完善的 CURD 操作及表达式能够满足大部分需求，但还不能完全代替传统的 SQL 语言，尤其在一些复杂的 SQL 查询中，所以系统也提供了传统的 SQL 语言查询。在 Model 模型中，有 3 种方式可以使用原生的 SQL 语言，下面分别进行介绍

1、query 方法（query 方法的使用比较简单和直观，接受的参数即为完整的 SQL 语句。这里所说的 SQL 语句是带完整数据表名的（包含前缀）查询语言。query 方法返回的结果是一个数据集，可以直接分配到模版中），2、execute 方法（execute 方法和 query 方法是一样的，使用方式也一样，唯一不同的是 execute 方法不会返回数据集，也不会返回受影响行数，甚至连错误信息都不返回。通常情况下 query 用于获取数据，execute 方法用于后台插入、更新数据。），3、exp 操作表达式（exp 操作表达式是介于 CURD 表达式与传统 SQL 之间的一种查询方式，为这两种方式找到了一个平衡点。在 exp 表达式中，开发人员可以完全使用标准的 SQL 语句（where 部分）进行数据操作，包括查询、更新、删除等）。

[执行原生 SQL 语句的方式](https://www.cnblogs.com/yunxintryyoubest/p/9906060.html)

...

原生 sql 语句：1，cursor 方法，2，row 方法：（掺杂着原生 sql 和 orm 来执行的操作，3，exclude 方法：（将什么排除）

...

在 orm 框架中，比如 hibernate 和 mybatis 都可以设置关联对象，比如 user 对象关联 dept
假如查询出 n 个 user，那么需要做 n 次查询 dept，查询 user 是一次 select，查询 user 关联的 dept，是 n 次，所以是 n+1 问题，其实叫**1+n 更为合理一些**。

1+n 问题是什么？应该怎样解决？

- 1+n 是执行一次查询获取 n 条主数据后，由于关联引起的执行 n 次查询从数据；它带来了性能问题；
- 一般来说，通过**懒加载** 可以部分缓解 1+n 带来的性能问题

SQL 中的 n+1 次 select 语句查询问题

如果当[SQL](http://database.51cto.com/art/201009/223949.htm)数据库中 select 语句数目过多，就会影响数据库的性能，如果需要查询 n 个 Customer 对象，那么必须执行 n+1 次 select 查询语句，下文就将为您讲解这个 n+1 次 select 查询问题。

在 Session 的缓存中存放的是相互关联的对象图。默认情况下，当 Hibernate 从数据库中加载 Customer 对象时，会同时加载所有关联的 Order 对象。以 Customer 和 Order 类为例，假定 ORDERS 表的 CUSTOMER_ID 外键允许为 null，图 1 列出了 CUSTOMERS 表和 ORDERS 表中的记录。

![img](http://images.51cto.com/files/uploadimg/20100903/1558560.gif)

以下 Session 的 find()方法用于到数据库中检索所有的 Customer 对象：

```
List customerLists=session.find("from Customer as c");
```

运行以上 find()方法时，Hibernate 将先查询 CUSTOMERS 表中所有的记录，然后根据每条记录的 ID，到 ORDERS 表中查询有参照关系的记录，Hibernate 将依次执行以下 select 语句：


```
select * from CUSTOMERS;
select * from ORDERS where CUSTOMER_ID=1;
select * from ORDERS where CUSTOMER_ID=2;
select * from ORDERS where CUSTOMER_ID=3;
select * from ORDERS where CUSTOMER_ID=4;
```

通过以上 5 条 `select` 语句，Hibernate 最后加载了 4 个 `Customer` 对象和 5 个 `Order` 对象，在内存中形成了一幅关联的对象图，参见图 2。

![img](http://images.51cto.com/files/uploading/20100903/1558561.gif)Hibernate 在检索与 `Customer` 关联的 `Order` 对象时，使用了默认的立即检索策略。这种检索策略存在两大不足：

(1) `select` 语句的数目太多，需要频繁的访问数据库，会影响检索性能。如果需要查询 n 个 `Customer` 对象，那么必须执行 $n+1$ 次 `select` 查询语句。这就是经典的 $n+1$ 次 `select` 查询问题。这种检索策略没有利用 SQL 的连接查询功能，例如以上 5 条 `select` 语句完全可以通过以下 1 条 `select` 语句来完成：

```
select * from CUSTOMERS left outer join ORDERS
on CUSTOMERS.ID=ORDERS.CUSTOMER_ID
```

以上 `select` 语句使用了 SQL 的左外连接查询功能，能够在一条 `select` 语句中查询出 `CUSTOMERS` 表的所有记录，以及匹配的 `ORDERS` 表的记录。

(2) 在应用逻辑只需要访问 `Customer` 对象，而不需要访问 `Order` 对象的场合，加载 `Order` 对象完全是多余的操作，这些多余的 `Order` 对象白白浪费了许多内存空间。

为了解决以上问题，Hibernate 提供了其他两种检索策略：延迟检索策略和迫切左外连接检索策略。延迟检索策略能避免多余加载应用程序不需要访问的关联对象，迫切左外连接检索策略则充分利用了 SQL 的外连接查询功能，能够减少 `select` 语句的数目。

ORM 框架使用优缺点

1. 什么是 ORM?

对象-关系映射 (Object-Relational Mapping, 简称 ORM)，面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系[数据库](http://lib.csdn.net/base/mysql)是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，对象-关系映射(ORM)系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

2. 为什么使用 ORM?

当我们实现一个应用程序时（不使用 O/R Mapping），我们可能会写特别多数据访问层的代码，从数据库保存、删除、读取对象信息，而这些代码都是重复的。而使用 ORM 则会大大减少重复性代码。对象关系映射 (Object Relational Mapping, 简称 ORM)，主要实现程序对象到关系数据库数据的映射。

3. 对象-关系映射解释:

A . 简单: ORM 以最基本的形式建模数据。比如 ORM 会将[MySQL](http://lib.csdn.net/base/mysql)的一张表映射成一个[Java](http://lib.csdn.net/base/javaee)类（模型），表的字段就是这个类的成员变量

- B . 精确: ORM 使所有的 MySQL 数据表都按照统一的标准精确地映射成 java 类, 使系统在代码层面保持准确统一
- C . 易懂: ORM 使数据库结构文档化。比如 MySQL 数据库就被 ORM 转换为了 java 程序员可以读懂的 java 类, java 程序员可以只把注意力放在他擅长的 java 层面 (当然能够熟练掌握 MySQL 更好)
- D. 易用: ORM 包含对持久类对象进行 CRUD 操作的 API, 例如 create(), update(), save(), load(), find(), find_all(), where()等, 也就是讲 sql 查询全部封装成了编程语言中的函数, 通过函数的链式组合生成最终的 SQL 语句。通过这种封装避免了不规范、冗余、风格不统一的 SQL 语句, 可以避免很多人为 Bug, 方便编码风格的统一和后期维护。

![这里写图片描述](https://img-blog.csdn.net/20160307142645476)

4. ORM 的优缺点:

优点:

- 1) 提高开发效率, 降低开发成本
- 2) 使开发更加对象化
- 3) 可移植
- 4) 可以很方便地引入数据缓存之类的附加功能

缺点:

- 1) 自动化进行关系数据库的映射需要消耗系统性能。其实这里的性能消耗还好啦, 一般来说都可以忽略之。
- 2) 在处理多表联查、where 条件复杂之类的查询时, ORM 的语法会变得复杂。

5. 常用框架:

...

- (1) Hibernate 全自动 需要写 hql 语句
- (2) iBATIS 半自动 自己写 sql 语句, 可操作性强, 小巧

...

ORM 实现 CRUD 操作数据库与模块的数据交互

在了解了 Django 提供的模型管理平台之后, 我们来看看如何从代码层面完成对模型的 CRUD (Create / Read / Update / Delete) 操作。我们可以通过 manage.py 开启 Shell 交互式环境, 然后使用 Django 内置的 ORM 框架对模型进行 CRUD 操作。

```
(venv)$ python manage.py shell
Python 3.6.4 (v3.6.4:d48ecea5, Dec 18 2017, 21:07:28)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
新增
from hrs.models import Dept, Emp
```

```
>>> dept = Dept(40, '研发 2 部', '深圳')
>>> dept.save()
>>> 更新
>>> dept.name = '研发 3 部'
>>> dept.save()
>>> 查询
>>> 查询所有对象。
>>> Dept.objects.all()
>>> <QuerySet [ <Dept: 研发 1 部>, <Dept: 销售 1 部>, <Dept: 运维 1 部>, <Dept: 研发 3 部>]>
>>> 过滤数据。
```

```

>>> Dept.objects.filter(name='研发3部') # 查询部门名称为“研发3部”的部门
>>> <QuerySet [<Dept: 研发3部>]>
>>>
>>> Dept.objects.filter(name__contains='研发') # 查询部门名称包含“研发”的部门(模糊查询)
>>> <QuerySet [<Dept: 研发1部>, <Dept: 研发3部>]>
>>>
>>> Dept.objects.filter(no__gt=10).filter(no__lt=40) # 查询部门编号大于10小于40的部门
>>> <QuerySet [<Dept: 销售1部>, <Dept: 运维1部>]>
>>>
>>> Dept.objects.filter(no__range=(10, 30)) # 查询部门编号在10到30之间的部门
>>> <QuerySet [<Dept: 研发1部>, <Dept: 销售1部>, <Dept: 运维1部>]>
>>>
>>> 查询单个对象。
>>> Dept.objects.get(pk=10)
>>> <Dept: 研发1部>
>>>
>>> Dept.objects.get(no=20)
>>> <Dept: 销售1部>
>>>
>>> Dept.objects.get(no__exact=30)
>>> <Dept: 运维1部>
>>>
>>> Dept.objects.filter(no=10).first()
>>> <Dept: 研发1部>
>>>
>>> 排序数据。
>>> Dept.objects.order_by('no') # 查询所有部门按部门编号升序排列
>>> <QuerySet [<Dept: 研发1部>, <Dept: 销售1部>, <Dept: 运维1部>, <Dept: 研发3部>]>
>>>
>>> Dept.objects.order_by('-no') # 查询所有部门按部门编号降序排列
>>> <QuerySet [<Dept: 研发3部>, <Dept: 运维1部>, <Dept: 销售1部>, <Dept: 研发1部>]>
>>> 数据切片(分页查询)。
>>> Dept.objects.order_by('no')[0:2] # 按部门编号排序查询1~2部门
>>> <QuerySet [<Dept: 研发1部>, <Dept: 销售1部>]>
>>>
>>> Dept.objects.order_by('no')[2:4] # 按部门编号排序查询3~4部门
>>> <QuerySet [<Dept: 运维1部>, <Dept: 研发3部>]>
>>> 高级查询。
>>> Emp.objects.filter(dept__no=10) # 根据部门编号查询该部门的员工
>>> <QuerySet [<Emp: 乔峰>, <Emp: 张无忌>, <Emp: 张三丰>]>
>>>
>>> Emp.objects.filter(dept__name__contains='销售') # 查询名字包含“销售”的部门的员工
>>> <QuerySet [<Emp: 黄蓉>]>
>>>
>>> Dept.objects.get(pk=10).emp_set.all() # 通过部门反查部门所有的员工
>>> <QuerySet [<Emp: 乔峰>, <Emp: 张无忌>, <Emp: 张三丰>]>

```

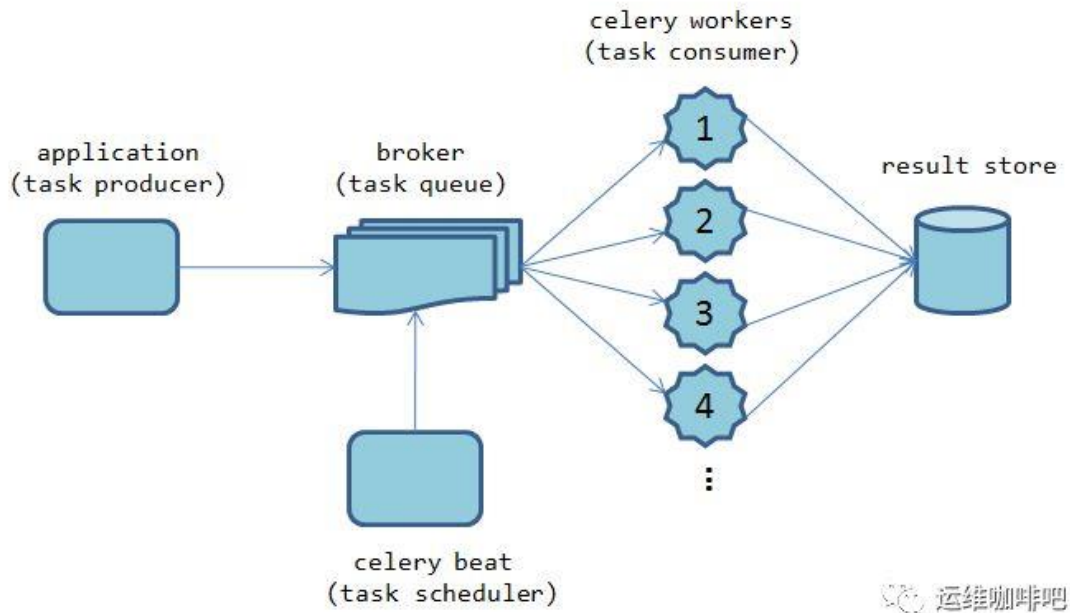
5. 如何执行异步任务和定时任务。

如何使用 [Django+Celery](#) 执行异步任务和定时任务

原生 Celery, 非 djcelery 模块, 所有演示均基于 Django2.0

celery 是一个基于 python 开发的简单、灵活且可靠的分布式任务队列框架，支持使用任务队列的方式在分布式的机器/进程/线程上执行任务调度。采用典型的生产者-消费者模型，主要由三部分组成：

6. 消息队列 broker: broker 实际上就是一个 MQ 队列服务，可以使用 Redis、RabbitMQ 等作为 broker
7. 处理任务的消费者 workers: broker 通知 worker 队列中有任务，worker 去队列中取出任务执行，每一个 worker 就是一个进程
8. 存储结果的 backend: 执行结果存储在 backend，默认也会存储在 broker 使用的 MQ 队列服务中，也可以单独配置用何种服务做 backend



【图片来自互联网】

异步任务

我的异步使用场景为项目上线：前端 web 上有个上线按钮，点击按钮后发请求给后端，后端执行上线过程要 5 分钟，后端在接收到请求后把任务放入队列异步执行，同时马上返回给前端一个任务执行中的结果。若果没有异步执行会怎么样呢？同步的情况就是执行过程中前端一直在等后端返回结果，页面转呀转的就转超时了。

异步任务配置

1.安装 RabbitMQ，这里我们使用 RabbitMQ 作为 broker，安装完成后默认启动了，也不需要其他任何配置

```
# apt-get install rabbitmq-server
```

2.安装 celery

```
# pip3 install celery
```

3.celery 用在 django 项目中，django 项目目录结构(简化)如下

```

website/
|-- deploy
|   |-- admin.py
|   |-- apps.py
|   |-- __init__.py
|   |-- models.py
|   |-- tasks.py
|   |-- tests.py
|   |-- urls.py
|   |-- views.py
|-- manage.py
|-- README
`-- website
    |-- celery.py
    |-- __init__.py
    |-- settings.py
    |-- urls.py
    |-- wsgi.py

```

4. 创建 website/celery.py 主文件

```

from __future__ import absolute_import, unicode_literals
import os
from celery import Celery, platforms

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'website.settings')

app = Celery('website')

# Using a string here means the worker don't have to serialize
# the configuration object to child processes.
# - namespace='CELERY' means all celery-related configuration keys
#   should have a `CELERY_` prefix.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Load task modules from all registered Django app configs.
app.autodiscover_tasks()

# 允许 root 用户运行 celery
platforms.C_FORCE_ROOT = True

@app.task(bind=True)
def debug_task(self):
    print('Request: {0!r}'.format(self.request))

```

5. 在 website/__init__.py 文件中增加如下内容，确保 django 启动的时候这个 app 能够被加载到

```

from __future__ import absolute_import

# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from .celery import app as celery_app

__all__ = ['celery_app']

```

6.各应用创建 tasks.py 文件，这里为 deploy/tasks.py

```
from __future__ import absolute_import
from celery import shared_task
```

```
@shared_task
def add(x, y):
    return x + y
```

- 注意 tasks.py 必须建在各 app 的根目录下，且只能叫 tasks.py，不能随意命名

7.views.py 中引用使用这个 tasks 异步处理

```
from deploy.tasks import add
```

```
def post(request):
    result = add.delay(2, 3)
```

- 使用函数名.delay()即可使函数异步执行
- 可以通过 result.ready()来判断任务是否完成处理
- 如果任务抛出一个异常，使用 result.get(timeout=1)可以重新抛出异常
- 如果任务抛出一个异常，使用 result.traceback 可以获取原始的回溯信息

8.启动 celery

```
# celery -A website worker -l info
```

9.这样在调用 post 这个方法时，里边的 add 就可以异步处理了

定时任务

定时任务的使用场景就很普遍了，比如我需要定时发送报告给老板~

定时任务配置

1.website/celery.py 文件添加如下配置以支持定时任务 crontab

```
from celery.schedules import crontab
```

```
app.conf.update(
    CELERYBEAT_SCHEDULE = {
        'sum-task': {
            'task': 'deploy.tasks.add',
            'schedule': timedelta(seconds=20),
            'args': (5, 6)
        }
        'send-report': {
            'task': 'deploy.tasks.report',
            'schedule': crontab(hour=4, minute=30, day_of_week=1),
        }
    }
)
```

- 定义了两个 task:
 - 名字为'sum-task'的 task, 每 20 秒执行一次 add 函数, 并传了两个参数 5 和 6
 - 名字为'send-report'的 task, 每周一早上 4: 30 执行 report 函数
- timedelta 是 datetime 中的一个对象, 需要 from datetime import timedelta 引入, 有如下几个参数
 - days: 天
 - seconds: 秒
 - microseconds: 微妙
 - milliseconds: 毫秒
 - minutes: 分
 - hours: 小时
- crontab 的参数有:
 - month_of_year: 月份
 - day_of_month: 日期
 - day_of_week: 周
 - hour: 小时
 - minute: 分钟

9. deploy/tasks.py 文件添加 report 方法:

```
@shared_task
def report():
    return 5
```

3.启动 celery beat, celery 启动了一个 beat 进程一直在不断的判断是否有任务需要执行

```
# celery -A website beat -l info
```

Tips

10. 如果你同时使用了异步任务和计划任务, 有一种更简单的启动方式 `celery -A website worker -b -l info`, 可同时启动 worker 和 beat
11. 如果使用的不是 rabbitmq 做队列那么需要在主配置文件中 `website/celery.py` 配置 broker 和 backend, 如下:

```
# redis 做 MQ 配置
app = Celery('website', backend='redis', broker='redis://localhost')
# rabbitmq 做 MQ 配置
app = Celery('website', backend='amqp', broker='amqp://admin:admin@localhost')
```

12. celery 不能用 root 用户启动的话需要在主配置文件中添加 `platforms.C_FORCE_ROOT = True`
13. celery 在长时间运行后可能出现内存泄漏，需要添加配置 `CELERYD_MAX_TASKS_PER_CHILD = 10`，表示每个 worker 执行了多少个任务就死掉

celery 执行异步任务和定时任务

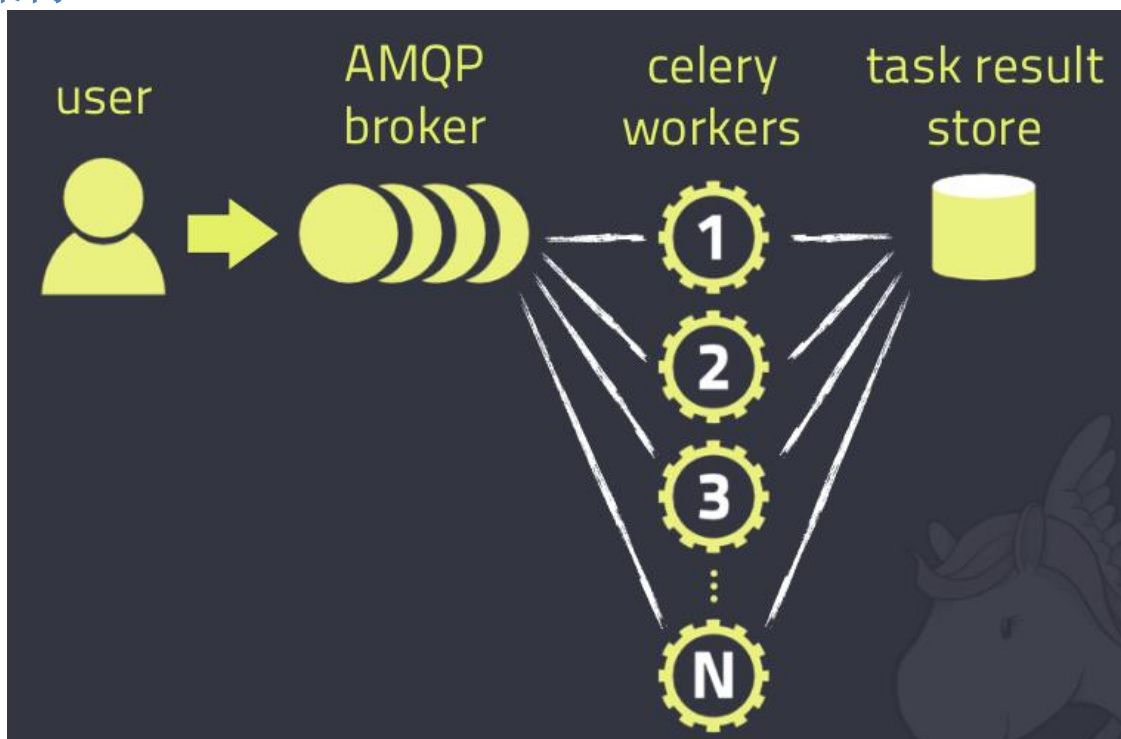
一、什么是 Celery

Celery 是一个简单、灵活且可靠的，处理大量消息的分布式系统

专注于实时处理的异步任务队列

同时也支持任务调度

Celery 架构



Celery 的架构由三部分组成，消息中间件（message broker），任务执行单元（worker）和任务执行结果存储（task result store）组成。

消息中间件

Celery 本身不提供消息服务，但是可以方便的和第三方提供的消息中间件集成。包括，RabbitMQ, Redis 等等

任务执行单元

Worker 是 Celery 提供的任务执行的单元，worker 并发的运行在分布式的系统节点中。

任务结果存储

Task result store 用来存储 Worker 执行的任务的结果，Celery 支持以不同方式存储任务的结果，包括 AMQP, redis 等

版本支持情况

Celery version 4.0 runs on

Python (2.7, 3.4, 3.5)

PyPy (5.4, 5.5)

This is the last version to support Python 2.7, and from the next version (Celery 5.x) Python 3.5 or newer is required.

If you're running an older version of Python, you need to be running an older version of Celery:

Python 2.6: Celery series 3.1 or earlier.

Python 2.5: Celery series 3.0 or earlier.

Python 2.4 was Celery series 2.2 or earlier.

Celery is a project with minimal funding, so we don't support Microsoft Windows. Please don't open any issues related to that platform.

二、使用场景

异步任务：将耗时操作任务提交给 Celery 去异步执行，比如发送短信/邮件、消息推送、音视频处理等等

定时任务：定时执行某件事情，比如每天数据统计

三、Celery 的安裝配置

pip install celery

消息中间件：RabbitMQ/Redis

app=Celery('任务名', backend='xxx',broker='xxx')

四、Celery 执行异步任务

基本使用

创建项目 celerytest

创建 py 文件：celeryapptask.py

```
import celery
import time
# broker='redis://127.0.0.1:6379/2' 不加密
backend='redis://:123456@127.0.0.1:6379/1'
broker='redis://:123456@127.0.0.1:6379/2'
cel=celery.Celery('test',backend=backend,broker=broker)
@cel.task
def add(x,y):
    return x+y
```

创建 py 文件：add_task.py,添加任务

```
from celery_app_task import add
result = add.delay(4,5)
print(result.id)
```

注：windows 下：celery worker -A celeryapptask -l info -P eventlet

```
from celery_app_task import cel
if __name__ == '__main__':
    cel.worker_main()
    # cel.worker_main(argv=['--loglevel=info'])
```

创建 py 文件：result.py，查看任务执行结果

```
from celery.result import AsyncResult
from celery_app_task import cel
async = AsyncResult(id="e919d97d-2938-4d0f-9265-fd8237dc2aa3", app=cel)
if async.successful():
    result = async.get()
    print(result)
    # result.forget() # 将结果删除
elif async.failed():
    print('执行失败')
elif async.status == 'PENDING':
    print('任务等待中被执行')
elif async.status == 'RETRY':
    print('任务异常后正在重试')
elif async.status == 'STARTED':
    print('任务已经开始被执行')
```

执行 add_task.py，添加任务，并获取任务 ID

执行 run.py，或者执行命令：celery worker -A celeryapptask -l info

执行 result.py,检查任务状态并获取结果

多任务结构

```
pro_cel
├── celery_task# celery 相关文件夹
│   ├── celery.py    # celery 连接和配置相关文件,必须叫这个名字
│   ├── tasks1.py    # 所有任务函数
│   └── tasks2.py    # 所有任务函数
├── check_result.py  # 检查结果
└── send_task.py    # 触发任务
```

celery.py

```
from celery import Celery
```

```
cel = Celery('celery_demo',
             broker='redis://127.0.0.1:6379/1',
             backend='redis://127.0.0.1:6379/2',
             # 包含以下两个任务文件，去相应的 py 文件中找任务，对多个任务做分类
             include=['celery_task.tasks1',
                     'celery_task.tasks2'
                     ])
```

```
# 时区
```

```
cel.conf.timezone = 'Asia/Shanghai'
```

```
# 是否使用 UTC
cel.conf.enable_utc = False
```

tasks1.py

```
import time
from celery_task.celery import cel
@cel.task
def test_celery(res):
    time.sleep(5)
    return "test_celery 任务结果:%s"%res
```

tasks2.py

```
import time
from celery_task.celery import cel
@cel.task
def test_celery2(res):
    time.sleep(5)
    return "test_celery2 任务结果:%s"%res
```

check_result.py

```
from celery.result import AsyncResult
from celery_task.celery import cel
async = AsyncResult(id="08eb2778-24e1-44e4-a54b-56990b3519ef", app=cel)
if async.successful():
    result = async.get()
    print(result)
    # result.forget() # 将结果删除,执行完成,结果不会自动删除
    # async.revoke(terminate=True) # 无论现在是什么时候,都要终止
    # async.revoke(terminate=False) # 如果任务还没有开始执行呢,那么就可以终止。
elif async.failed():
    print('执行失败')
elif async.status == 'PENDING':
    print('任务等待中被执行')
elif async.status == 'RETRY':
    print('任务异常后正在重试')
elif async.status == 'STARTED':
    print('任务已经开始被执行')
```

send_task.py

```
from celery_task.tasks1 import test_celery
from celery_task.tasks2 import test_celery2
# 立即告知 celery 去执行 test_celery 任务,并传入一个参数
result = test_celery.delay('第一个的执行')
print(result.id)
result = test_celery2.delay('第二个的执行')
print(result.id)
```

添加任务（执行 `sendtask.py`），开启 `work: celery worker -A celerytask -l info -P eventlet`，检查任务执行结果（执行 `check_result.py`）

五、Celery 执行定时任务

设定时间让 celery 执行一个任务

add_task.py

```
from celery_app_task import add
from datetime import datetime
# 方式一
# v1 = datetime(2019, 2, 13, 18, 19, 56)
# print(v1)
# v2 = datetime.utcnow().timestamp(v1.timestamp())
# print(v2)
# result = add.apply_async(args=[1, 3], eta=v2)
# print(result.id)
# 方式二
ctime = datetime.now()
# 默认用 utc 时间
utc_ctime = datetime.utcnow().timestamp(ctime.timestamp())
from datetime import timedelta
time_delay = timedelta(seconds=10)
task_time = utc_ctime + time_delay

# 使用 apply_async 并设定时间
result = add.apply_async(args=[4, 3], eta=task_time)
print(result.id)
```

类似于 `crontab` 的定时任务

多任务结构中 celery.py 修改如下

```
from datetime import timedelta
from celery import Celery
from celery.schedules import crontab

cel = Celery('tasks', broker='redis://127.0.0.1:6379/1', backend='redis://127.0.0.1:6379/2', include=[
    'celery_task.tasks1',
    'celery_task.tasks2',
])
cel.conf.timezone = 'Asia/Shanghai'
cel.conf.enable_utc = False

cel.conf.beat_schedule = {
    # 名字随意命名
    'add-every-10-seconds': {
        # 执行 tasks1 下的 test_celery 函数
        'task': 'celery_task.tasks1.test_celery',
        # 每隔 2 秒执行一次
        # 'schedule': 1.0,
        # 'schedule': crontab(minute="*/1"),
        'schedule': timedelta(seconds=2),
        # 传递参数
        'args': ('test',)
    },
    # 'add-every-12-seconds': {
    #     'task': 'celery_task.tasks1.test_celery',
```

```

# 每年 4 月 11 号, 8 点 42 分执行
# 'schedule': crontab(minute=42, hour=8, day_of_month=11, month_of_year=4),
# 'schedule': crontab(minute=42, hour=8, day_of_month=11, month_of_year=4),
# 'args': (16, 16)
# },
}

```

启动一个 beat: `celery beat -A celery_task -l info`

启动 work 执行: `celery worker -A celery_task -l info -P eventlet`

六、Django 中使用 Celery

在项目目录下创建 `celeryconfig.py`

```

import djcelery
djcelery.setup_loader()
CELERY_IMPORTS=(
    'app01.tasks',
)
#有些情况可以防止死锁
CELERYD_FORCE_EXECV=True
# 设置并发 worker 数量
CELERYD_CONCURRENCY=4
#允许重试
CELERY_ACKS_LATE=True
# 每个 worker 最多执行 100 个任务被销毁, 可以防止内存泄漏
CELERYD_MAX_TASKS_PER_CHILD=100
# 超时时间
CELERYD_TASK_TIME_LIMIT=12*30

```

在 `app01` 目录下创建 `tasks.py`

```

from celery import task
@task
def add(a,b):

from django.shortcuts import render,HttpResponse
from app01.tasks import add
from datetime import datetime
def test(request):
    # result=add.delay(2,3)
    ctime = datetime.now()
    # 默认用 utc 时间
    utc_ctime = datetime.utcnow().timestamp()
    from datetime import timedelta
    time_delay = timedelta(seconds=5)
    task_time = utc_ctime + time_delay
    result = add.apply_async(args=[4, 3], eta=task_time)
    print(result.id)
    return HttpResponse('ok')

```

`settings.py`

```

with open('a.text', 'a', encoding='utf-8') as f:
    f.write('a')
print(a+b)

```

视图函数 views.py

```
INSTALLED_APPS = [
    ...
    'djcelery',
    'app01'
]

...

from djagocele import celeryconfig
BROKER_BACKEND='redis'
BOOKER_URL='redis://127.0.0.1:6379/1'
CELERY_RESULT_BACKEND='redis://127.0.0.1:6379/2'
```

6. 如何实现页面缓存和查询缓存？缓存如何预热？

Django 中提供了 6 种缓存方式

- 开发调试
- 内存
- 文件
- 数据库
- Memcache 缓存（python-memcached 模块）
- Memcache 缓存（pylibmc 模块）

Redis（配合 Django 使用）

Django + Redis 实现页面缓存

目的：把从数据库读出的数据存入的 redis 中既提高了效率，又减少了对数据库的读写，提高用户体验。

例如：

1，同一页面局部缓存，局部动态

```
from django.views import View
from myapp.models import Student
#导入缓存库
from django.core.cache import cache
#导入页面缓存
from django.views.decorators.cache import cache_page
from django.utils.decorators import method_decorator
class Stulist(View):
    def get(self,request,id):
        #判断缓存内是否有数据
        result = cache.get("res",'0')
        if result == '0':
            res = Student.objects.filter(id=id)
            cache.set("res",res,100)
            result =cache.get("res")
        # ret = Student.objects.all()
```

```

# ret = [i.name for i in list(ret)]
# random_name = random.sample(ret,3)
#随机取一条 select * from student where id in(2,3) order by rand limit 1
#取非当前数据外三条数据随机展示
random_name = Student.objects.exclude(id__in=[id]).order_by("?")[0:3]
return render(request,'stulist.html',locals())

```

2, 页面缓存

```

@cache_page(60)
def page_cache(request):
    res = Student.objects.all()
    return render(request,'pagecache.html',locals())
@method_decorator(cache_page(60),name="get")
class PageCache(View):
    def get(self,request):
        res = Student.objects.all()

        return render(request,'pagecache.html',locals())

```

2.1 各种缓存方式的配置文件说明

2.1.1 开发调试(此模式为开发调试使用,实际上不执行任何操作)

settings.py 文件配置

```

CACHES = {'default': {'BACKEND': 'django.core.cache.backends.dummy.DummyCache', # 缓存后台使用的引擎'TIMEOUT': 300, # 缓存超时时间(默认 300 秒, None 表示永不过期, 0 表示立即过期)'OPTIONS': {'MAX_ENTRIES': 300, # 最大缓存记录的数量(默认 300)'CULL_FREQUENCY': 3, # 缓存到达最大个数之后, 剔除缓存个数的比例, 即: 1/CULL_FREQUENCY(默认 3)},}}

```

2.1.2 内存缓存(将缓存内容保存至内存区域中)

settings.py 文件配置

```

CACHES = {'default': {'BACKEND': 'django.core.cache.backends.locmem.LocMemCache', # 指定缓存使用的引擎'LOCATION': 'unique-snowflake', # 写在内存中的变量的唯一值'TIMEOUT':300, # 缓存超时时间(默认为 300 秒,None 表示永不过期)'OPTIONS':{'MAX_ENTRIES': 300, # 最大缓存记录的数量(默认 300)'CULL_FREQUENCY': 3, # 缓存到达最大个数之后, 剔除缓存个数的比例, 即: 1/CULL_FREQUENCY(默认 3)}}}

```

2.1.3 文件缓存(把缓存数据存储在文件中)

settings.py 文件配置

```

CACHES = {'default': {'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache', #指定缓存使用的引擎'LOCATION': '/var/tmp/django_cache', #指定缓存的路径'TIMEOUT':300, #缓存超时时间(默认为 300 秒,None 表示永不过期)'OPTIONS':{'MAX_ENTRIES': 300, # 最大缓存记录的数量(默认 300)'CULL_FREQUENCY': 3, # 缓存到达最大个数之后, 剔除缓存个数的比例, 即: 1/CULL_FREQUENCY(默认 3)}}}

```

2.1.4 数据库缓存(把缓存数据存储在数据库中)

settings.py 文件配置

```

CACHES = {'default': {'BACKEND': 'django.core.cache.backends.db.DatabaseCache', # 指定缓存使用的引擎'LOCATION': 'cache_table', # 数据库表'OPTIONS':{'MAX_ENTRIES': 300, # 最大缓存记录的数量

```

量（默认 300）'CULL_FREQUENCY': 3, # 缓存到达最大个数之后，剔除缓存个数的比例，即：1/CULL_FREQUENCY（默认 3）}}}

注意,创建缓存的数据库表使用的语句:

```
python manage.py createcachetable
```

Memcached 是 Django 原生支持的缓存系统.要使用 Memcached,需要下载 Memcached 的支持库 python-memcached 或 pylibmc.

2.1.5 Memcache 缓存(使用 python-memcached 模块连接 memcache)

settings.py 文件配置

```
CACHES = {'default': {'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache', # 指定缓存使用的引擎'LOCATION': '192.168.10.100:11211', # 指定 Memcache 缓存服务器的 IP 地址和端口'OPTIONS': {'MAX_ENTRIES': 300, # 最大缓存记录的数量（默认 300）'CULL_FREQUENCY': 3, # 缓存到达最大个数之后，剔除缓存个数的比例，即：1/CULL_FREQUENCY（默认 3）}}}
```

LOCATION 也可以配置成如下:

```
'LOCATION': 'unix:/tmp/memcached.sock', # 指定局域网内的主机名加 socket 套接字为 Memcache 缓存服务器'LOCATION': [ # 指定一台或多台其他主机 ip 地址加端口为 Memcache 缓存服务器'192.168.10.100:11211', '192.168.10.101:11211', '192.168.10.102:11211', ]
```

2.1.6 Memcache 缓存(使用 pylibmc 模块连接 memcache)

```
settings.py 文件配置 CACHES = {'default': {'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache', # 指定缓存使用的引擎'LOCATION': '192.168.10.100:11211', # 指定本机的 11211 端口为 Memcache 缓存服务器'OPTIONS': {'MAX_ENTRIES': 300, # 最大缓存记录的数量（默认 300）'CULL_FREQUENCY': 3, # 缓存到达最大个数之后，剔除缓存个数的比例，即：1/CULL_FREQUENCY（默认 3）}}, }
```

LOCATION 也可以配置成如下:

```
'LOCATION': '/tmp/memcached.sock', # 指定某个路径为缓存目录'LOCATION': [ # 分布式缓存,在多台服务器上运行 Memcached 进程,程序会把多台服务器当作一个单独的缓存,而不会在每台服务器上复制缓存值'192.168.10.100:11211', '192.168.10.101:11211', '192.168.10.102:11211', ]
```

Memcached 是基于内存的缓存,数据存储在内存中.所以如果服务器死机的话,数据就会丢失,所以 Memcached 一般与其他缓存配合使用

3.Django 中的缓存应用

Django 提供了不同粒度的缓存,可以缓存某个页面,可以只缓存一个页面的某个部分,甚至可以缓存整个网站.

3.1 单独视图缓存

例子,为单个视图函数添加缓存

路由配置:

```
url(r'^index$', views.index),
```

数据库

<Filter criteria>			
	id	name	pwd
1	1	python	python123
2	4	openstack	openstack123
3	2	linux	linux123
4	3	golang	golang123

views 代码:

```
from app01 import models
from django.views.decorators.cache import cache_page
import time

@cache_page(15) #超时时间为 15 秒
def index(request):
    user_list=models.UserInfo.objects.all() #从数据库中取出所有的用户对象
    ctime=time.time() #获取当前时间
    return render(request,"index.html",{"user_list":user_list,"ctime":ctime})
```

index.html 代码:

```
body><h1>{{ ctime }}</h1><ul>{% for user in user_list %}<li>{{ user.name }}</li>{% endfor %}</ul></body>
```

因为缓存的原因,不停的刷新浏览器时会发现,页面上显示的时间每 15 秒钟变化一次.

在立即刷新浏览器的时候,立即在数据库中添加一个用户对象,此时继续刷新浏览器,前端页面上不会显示刚才添加的用户

一直刷新浏览器 15 秒后,新添加的用户才用在前端页面上显示出来.

上面的例子是基于内存的缓存配置,基于文件的缓存该怎么配置呢??

更改 settings.py 的配置

```
CACHES = {'default': {'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache', # 指定缓存使用的引擎
                      'LOCATION': 'E:\django_cache', # 指定缓存的路径
                      'TIMEOUT': 300, # 缓存超时时间 (默认为 300 秒,None 表示永不过期)
                      'OPTIONS': {'MAX_ENTRIES': 300, # 最大缓存记录的数量 (默认 300)
                                   'CULL_FREQUENCY': 3, # 缓存到达最大个数之后,剔除缓存个数的比例,即: 1/CULL_FREQUENCY (默认 3)
                                }}}}
```

然后再次刷新浏览器,可以看到在刚才配置的目录下生成的缓存文件

计算机 > 文档 (E:) > django_cache				
包含到库中 共享 新建文件夹				
名称	修改日期	类型	大小	
6f134a4214467b7ea4185c595222780...	2017/9/20 19:02	DJCACHE 文件	1 KB	
f97acf8fbbde79f7616458c704836ade....	2017/9/20 19:02	DJCACHE 文件	1 KB	

通过实验可以知道,Django 会以自己的形式把缓存文件保存在配置文件中指定的目录中.

3.2 全站使用缓存

既然是全站缓存,当然要使用 Django 中的中间件.

用户的请求通过中间件,经过一系列的认证等操作,如果请求的内容在缓存中存在,则使用 FetchFromCacheMiddleware 获取内容并返回给用户

当返回给用户之前,判断缓存中是否已经存在,如果不存在,则 UpdateCacheMiddleware 会将缓存保存至 Django 的缓存之中,以实现全站缓存

修改 settings.py 配置文件

```
MIDDLEWARE = ['django.middleware.cache.UpdateCacheMiddleware', #响应 HttpResponse 中设置几个 headers
'django.middleware.security.SecurityMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'django.middleware.clickjacking.XFrameOptionsMiddleware', 'django.middleware.cache.FetchFromCacheMiddleware', #用来缓存通过 GET 和 HEAD 方法获取的状态码为 200 的响应]
CACHE__MIDDLEWARE_SECONDS=15 # 设定超时时间为 15 秒
```

views 视图函数

```
from django.shortcuts import renderimport time def index(request):ctime = time.time()return render(request, 'index.html', {'ctime': ctime})
```

其余代码不变,刷新浏览器是 15 秒,页面上的时间变化一次,这样就实现了全站缓存.

3.3 局部视图缓存

例子,刷新页面时,整个网页有一部分实现缓存

views 视图函数

```
from django.shortcuts import renderimport time def index(request):# user_list = models.UserInfo.objects.all()ctime = time.time()return render(request, 'index.html', {'ctime': ctime})
```

前端网页

```
{% load cache %} # 加载缓存<!DOCTYPE html><html lang="en"><head><meta charset="UTF-8"><title>Title</title></head><body><h1>{{ ctime }}</h1>{% cache 15 'aaa' %} # 设定超时时间为 15 秒<h1>{{ ctime }}</h1>{% endcache %}</body></html>
```

刷新浏览器可以看到,第一个时间实时变化,后面一个时间每 15 秒钟变化一次

Django 项目之使用缓存进行数据查询

省市区的数据是经常被用户查询使用的,而且数据基本不变化,所以我们可以将省市区数据进行缓存处理,减少数据库的查询次数。

在 Django REST framework 中使用缓存,可以通过 drf-extensions 扩展来实现。

关于扩展使用缓存的文档,可参考链接 <http://chibisov.github.io/drf-extensions/docs/#caching>

安装

pip install drf-extensions

1

使用方法

1) 直接添加装饰器

可以在使用 `restframeworkextensions.cache.decorators` 中的 `cache_response` 装饰器来装饰返回数据的类视图的对象方法,如

```
class CityView(views.APIView):
    @cacheresponse()
    def get(self, request, *args, **kwargs):
```

...

`cacheresponse` 装饰器可以接收两个参数

`@cacheresponse(timeout=60*60, cache='default')`

`timeout` 缓存时间

`cache` 缓存使用的 Django 缓存后端（即 `CACHES` 配置中的键名称）

如果在使用 `cacheresponse` 装饰器时未指明 `timeout` 或者 `cache` 参数，则会使用配置文件中的默认配置，可以通过如下方法指明：

DRF 扩展

```
RESTFRAMEWORKEXTENSIONS = {
```

```
# 缓存时间
```

```
'DEFAULTCACHERESPONSETIMEOUT': 60 * 60,
```

```
# 缓存存储
```

```
'DEFAULTUSECACHE': 'default',
```

```
}
```

`DEFAULTCACHERESPONSETIMEOUT` 缓存有效期，单位秒

`DEFAULTUSECACHE` 缓存的存储方式，与配置文件中的 `CACHES` 的键对应。

注意：

1. `cache_response` 装饰器既可以装饰在类视图中的 `get` 方法上，

2. 也可以装饰在 REST framework 扩展类提供的 `list` 或 `retrieve` 方法上。

3. 使用 `cacheresponse` 装饰器无需使用 `methoddecorator` 进行转换。

2) 使用 `drf-extensions` 提供的扩展类

`drf-extensions` 扩展对于缓存提供了三个扩展类：

`ListCacheResponseMixin`

用于缓存返回列表数据的视图，与 `ListModelMixin` 扩展类配合使用，实际是为 `list` 方法添加了 `cache_response` 装饰器

`RetrieveCacheResponseMixin`

用于缓存返回单一数据的视图，与 `RetrieveModelMixin` 扩展类配合使用，实际是为 `retrieve` 方法添加了 `cache_response` 装饰器

`CacheResponseMixin`

为视图集同时补充 `List` 和 `Retrieve` 两种缓存，与 `ListModelMixin` 和 `RetrieveModelMixin` 一起配合使用。

三个扩展类都是在 `restframeworkextensions.cache.mixins` 中。

缓存数据保存位置与有效期的设置

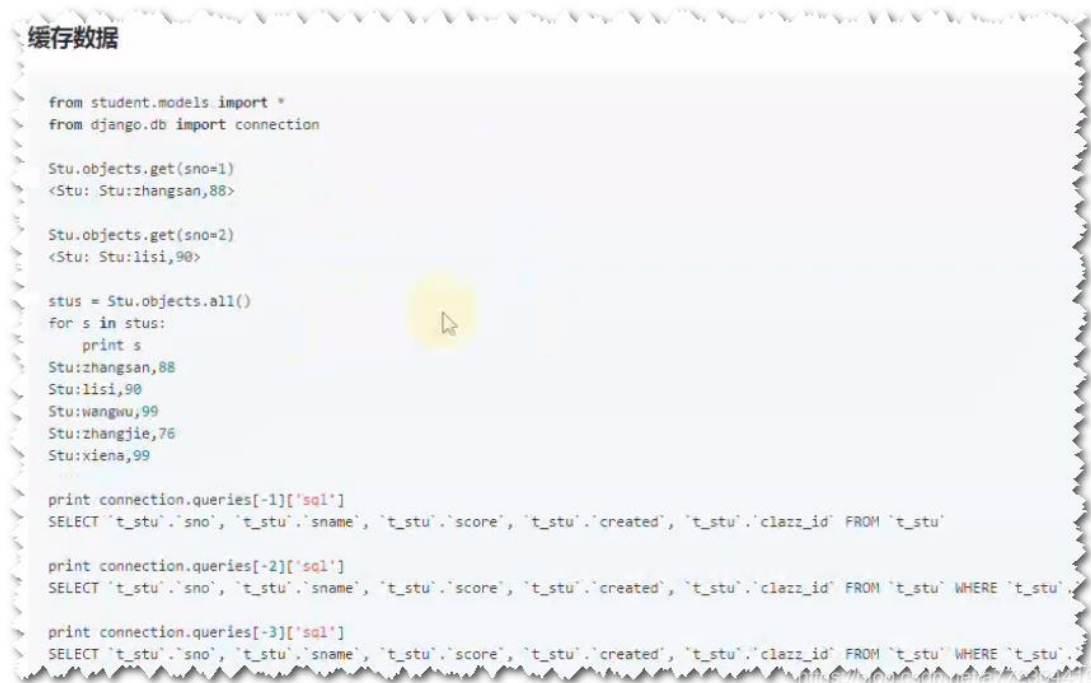
我们想把缓存数据保存在 `redis` 中，且设置有效期，可以通过在配置文件中定义的方式来实现。

在配置文件中增加

DRF 扩展

```
RESTFRAMEWORKEXTENSIONS = {  
    ##### 缓存时间  
    'DEFAULTCACHERESPONSETIMEOUT': 60 * 60,  
    ##### 缓存存储 default 表示缓存设置中设置的默认存储位置  
    'DEFAULTUSE_CACHE': 'default',  
}
```

Python Django 查询数据缓存



缓存刷新、缓存预热的区别和使用场景是什么？

缓存刷新：强制将分发节点上缓存的资源标记为过期，当用户再次对该资源发起请求时，节点会回源拉取资源，并缓存一份更新后的资源在分发节点

缓存预热：主动触发将源站资源推送到边缘节点，用户访问资源时，可以直接命中缓存，缓解突增回源流量给源站造成的压力

一、缓存预热

在刚启动的缓存系统中，如果缓存中没有任何数据，如果依靠用户请求的方式重建缓存数据，那么对数据库的压力非常大，而且系统的性能开销也是巨大的。

此时，最好的策略是启动时就把热点数据加载好。这样，用户请求时，直接读取的就是缓存的数据，而无需去读取 DB 重建缓存数据。

举个例子，热门的或者推荐的商品，需要提前预热到缓存中。

一般来说，有如下几种方式来实现：

数据量不大时，项目启动时，自动进行初始化。

写个修复数据脚本，手动执行该脚本。

写个管理界面，可以手动点击，预热对应的数据到缓存中。

二、缓存数据的淘汰策略

除了缓存服务器自带的缓存自动失效策略之外，我们还可以根据具体的业务需求进行自定义的“手动”缓存淘汰，常见的策略有两种：

1、定时去清理过期的缓存。

2、当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！

三、缓存如何存储 POJO 对象

实际场景下，缓存值可能是一个 POJO 对象，就需要考虑如何 POJO 对象存储的问题。目前有两种方式：

方案一，将 POJO 对象序列化进行存储，适合 Redis 和 Memcached。

当我们的数据存储在 Redis 的时候，我们的键（key）和值（value）都是通过 Spring 提供的 Serializer 序列化到数据库的。RedisTemplate 默认使用的是 JdkSerializationRedisSerializer，StringRedisTemplate 默认使用的是 StringRedisSerializer。

Spring Data JPA 为我们提供了下面的 Serializer：GenericToStringSerializer、Jackson2JsonRedisSerializer、JacksonJsonRedisSerializer、JdkSerializationRedisSerializer、OxmSerializer、StringRedisSerializer。

序列化方式对比：

JdkSerializationRedisSerializer: 使用 JDK 提供的序列化功能。优点是反序列化时不需要提供类型信息（class），但缺点是需要实现 Serializable 接口，还有序列化后的结果非常庞大，是 JSON 格式的 5 倍左右，这样就会消耗 redis 服务器的大量内存。

Jackson2JsonRedisSerializer: 使用 Jackson 库将对象序列化为 JSON 字符串。优点是速度快，序列化后的字符串短小精悍，不需要实现 Serializable 接口。但缺点也非常致命，那就是此类的构造函数中有一个类型参数，必须提供要序列化对象的类型信息（.class 对象）。通过查看源代码，发现其只在反序列化过程中用到了类型信息。

方案二，使用 Hash 数据结构，适合 Redis。

一、缓存穿透（击穿）

原理：缓存穿透（击穿）是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，数据库的压力骤增（可能会宕机）。

解决方法：

1. 布隆过滤器

对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃。还有最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

2. 缓存空对象

0 如果一个查询返回的数据为空（无论是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间（如果是攻击，问题更严重），比较有效的方法是针对这类数据设置一个较短的过期时间（最长不超过五分钟），让其自动剔除。

第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5 分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

二、缓存雪崩

原理：缓存雪崩是缓存时集中在某一时段同时失效，请求全部转发到数据库，数据库瞬时压力过重导致雪崩效应。

解决方法：

1. 加锁排队

在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。

2. 缓存时间增加随机值

在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

三、缓存预热

新的缓存系统没有任何缓存数据，在缓存重建数据的过程中，系统性能和数据库负载都不太好，所以最好是在系统上线之前就把要缓存的热点数据加载到缓存中，这种缓存预加载手段就是缓存预热。

四、缓存热备

缓存热备即当一台缓存服务器不可用时能实时切换到备用缓存服务器，不影响缓存使用。集群模式下，每个主节点都会有一个或多个从节点来当备用，一旦主节点挂点，从节点立即充当主节点使用。

缓存预热就是系统上线后，提前将相关的缓存数据加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题，用户直接查询事先被预热的缓存数据。

缓存预热解决方案：直接写个缓存刷新页面，上线时手工操作下；数据量不大，可以在项目启动的时候自动进行加载；定时刷新缓存。

爬虫相关

1. Scrapy 框架的组件和数据处理流程。
2. 爬取的目的（项目中哪些地方需要用到爬虫的数据）。
3. 使用的工具（抓包、c 下载、清理、存储、分析、可视化）。
4. 数据的来源（能够轻松的列举出 10 个网站）。
5. 数据的构成（抓取的某个字段在项目中有什么用）。

6. 反反爬措施（限速、请求头、Cookie 池、代理池、Selenium、PhantomJS、RoboBrowser、TOR、OCR）。
7. 数据的体量（最后抓取了多少数据，多少 W 条数据或多少个 G 的数据）。
8. 后期数据处理（持久化、数据补全、归一化、格式化、转存、分类）。

爬虫相关：

1、 Scrapy 框架的组件和数据处理流程。

2、 爬取的目的（项目中哪些地方需要用到爬虫的数据）。

0、IT 桔子和 36Kr 在专栏文章中（<http://zhuanlan.zhihu.com/p/20714713>），抓取 IT 橘子和 36Kr 的各公司的投融资数据，试图分析中国各家基金之间的互动关系。

1、知乎沧海横流，看行业起伏，抓取并汇总所有的答案，方便大家阅读，找出 2015 年最热门和最衰落的行业。

2、汽车之家大数据画像：宝马车主究竟有多任性？利用论坛发言的抓取以及 NLP，对各种车型的车主做画像。

3、天猫、京东、淘宝等电商网站超越咨询顾问的算力，在用户理解和维护，抓取各大电商的评论及销量数据，对各种商品（颗粒度可到款式）沿时间序列的销量以及用户的消费场景进行分析。甚至还可以根据用户评价做情感分析，实时监控产品在消费者心目中的形象，对新发布的产品及时监控，以便调整策略。

4、58 同城的房产、安居客、Q 房网、搜房等房产网站下半年深圳房价将如何发展，抓取房产买卖及租售信息，对热热闹闹的房价问题进行分析。

5、大众点评、美团网等餐饮及消费类网站黄焖鸡米饭是怎么火起来的？抓取各种店面的开业情况以及用户消费和评价，了解周边变化的口味，所谓是“舌尖上的爬虫”。以及各种变化的口味，比如：啤酒在衰退，重庆小面在崛起。

6、58 同城等分类信息网站花 10 万买贡茶配方，贵不贵？抓取招商加盟的数据，对定价进行分析，帮助网友解惑。

7、拉勾网、中华英才网等招聘网站互联网行业哪个职位比较有前途？抓取各类职位信息，分析最热门的职位以及薪水。

8、挂号网等医疗信息网站如何评价挂号网？抓取医生信息并于宏观情况进行交叉对比。

9、应用宝等 App 市场你用 Python 做过什么有趣的数据挖掘/分析项目？对各个 App 的发展情况进行跟踪及预测。（顺便吹一下牛，我们这个榜单很早就发现小红书 App 的快速增长趋势以及在年轻人中的极佳口碑）

10、携程、去哪儿及 12306 等交通出行类网站，对航班及高铁等信息进行抓取，能从一个侧面反映经济是否正在走入下行通道。

11、雪球等财经类网站抓取雪球 KOL 或者高回报用户的行为，找出推荐股票

12、58 同城二手车、易车等汽车类网站一年当中买车的最佳时间为何时？什么品牌或者型号的二手车残值高？更保值？反之，什么类型的贬值较快？ - 二手车，找出最佳的买车时间以及最保值的汽车。

13、神州租车、一嗨租车等租车类网站抓取它们列举出来的租车信息，长期跟踪租车价格及数量等信息

14、各类信托网站通过抓取信托的数据，了解信托项目的类型及规模

3、 使用的工具（抓包、c 下载、清理、存储、分析、可视化）。

网络

通用

urllib -网络库(stdlib)。

requests -网络库。

grab - 网络库（基于 pycurl）。

pycurl - 网络库（绑定 libcurl）。

urllib3 - Python HTTP 库，安全连接池、支持文件 post、可用性高。

httplib2 - 网络库。

RoboBrowser - 一个简单的、极具 Python 风格的 Python 库，无需独立的浏览器即可浏览网页。

MechanicalSoup - 一个与网站自动交互 Python 库。

mechanize - 有状态、可编程的 Web 浏览库。

socket - 底层网络接口(stdlib)。

Unirest for Python - Unirest 是一套可用于多种语言的轻量级的 HTTP 库。

hyper - Python 的 HTTP/2 客户端。

PySocks - SocksiPy 更新并积极维护的版本，包括错误修复和一些其他的特征。作为 socket 模块的直接替换。

异步

treq - 类似于 requests 的 API（基于 twisted）。

aiohttp - asyncio 的 HTTP 客户端/服务器(PEP-3156)。

网络爬虫框架

功能齐全的爬虫

grab - 网络爬虫框架（基于 pycurl/multicur）。

scrapy - 网络爬虫框架（基于 twisted），不支持 Python3。

pyspider - 一个强大的爬虫系统。

cola - 一个分布式爬虫框架。

其他

portia - 基于 Scrapy 的可视化爬虫。

restkit - Python 的 HTTP 资源工具包。它可以让你轻松地访问 HTTP 资源，并围绕它建立的对象。

demiurge - 基于 PyQuery 的爬虫微框架。

HTML/XML 解析器

通用

lxml - C 语言编写高效 HTML/ XML 处理库。支持 XPath。

cssselect - 解析 DOM 树和 CSS 选择器。

pyquery - 解析 DOM 树和 jQuery 选择器。

BeautifulSoup - 低效 HTML/ XML 处理库，纯 Python 实现。

html5lib - 根据 WHATWG 规范生成 HTML/ XML 文档的 DOM。该规范被用在现在所有的浏览器上。

feedparser - 解析 RSS/ATOM feeds。

MarkupSafe - 为 XML/HTML/XHTML 提供了安全转义的字符串。

xmldict - 一个可以让你在处理 XML 时感觉像在处理 JSON 一样的 Python 模块。

xhtml2pdf - 将 HTML/CSS 转换为 PDF。

untangle - 轻松实现将 XML 文件转换为 Python 对象。

清理

Bleach – 清理 HTML（需要 **html5lib**）。

sanitize – 为混乱的数据世界带来清明。

文本处理

用于解析和操作简单文本的库。

通用

difflib –（Python 标准库）帮助进行差异化比较。

Levenshtein – 快速计算 **Levenshtein** 距离和字符串相似度。

fuzzywuzzy – 模糊字符串匹配。

esmere – 正则表达式加速器。

ftfy – 自动整理 Unicode 文本，减少碎片化。

转换

unicode – 将 Unicode 文本转为 ASCII。

字符编码

uniout – 打印可读字符，而不是被转义的字符串。

chardet – 兼容 Python 的 2/3 的字符编码器。

xpinyin – 一个将中国汉字转为拼音的库。

pangu.py – 格式化文本中 CJK 和字母数字的间距。

Slug 化

awesome-slugify – 一个可以保留 unicode 的 Python slugify 库。

python-slugify – 一个可以将 Unicode 转为 ASCII 的 Python slugify 库。

unicode-slugify – 一个可以将生成 Unicode slugs 的工具。

pytils – 处理俄语字符串的简单工具（包括 **pytils.translit.slugify**）。

通用解析器

PLY – **lex** 和 **yacc** 解析工具的 Python 实现。

pyparsing – 一个通用框架的生成语法分析器。

人的名字

python-nameparser -解析人的名字的组件。

电话号码

phonenumbers -解析，格式化，存储和验证国际电话号码。

用户代理字符串

python-user-agents – 浏览器用户代理的解析器。

HTTP Agent Parser – Python 的 HTTP 代理分析器。

特定格式文件处理

解析和处理特定文本格式的库。

通用

tablib – 一个把数据导出为 XLS、CSV、JSON、YAML 等格式的模块。

textract – 从各种文件中提取文本，比如 Word、PowerPoint、PDF 等。

messytables – 解析混乱的表格数据的工具。

rows – 一个常用数据接口，支持的格式很多（目前支持 CSV，HTML，XLS，TXT – 将来还会提供更多！）。

Office

python-docx – 读取，查询和修改的 Microsoft Word2007/2008 的 docx 文件。

xlwt / xlrd – 从 Excel 文件读取写入数据和格式信息。

XlsxWriter – 一个创建 Excel.xlsx 文件的 Python 模块。

xlwings – 一个 BSD 许可的库，可以很容易地在 Excel 中调用 Python，反之亦然。

openpyxl – 一个用于读取和写入的 Excel2010 XLSX/ XLSM/ xlsx/ XLTM 文件的库。

Marmir – 提取 Python 数据结构并将其转换为电子表格。

PDF

PDFMiner – 一个从 PDF 文档中提取信息的工具。

PyPDF2 – 一个能够分割、合并和转换 PDF 页面的库。

ReportLab – 允许快速创建丰富的 PDF 文档。

pdftables – 直接从 PDF 文件中提取表格。

Markdown

Python-Markdown – 一个用 Python 实现的 John Gruber 的 Markdown。

Mistune – 速度最快，功能全面的 Markdown 纯 Python 解析器。

markdown2 – 一个完全用 Python 实现的快速的 Markdown。

YAML

PyYAML – 一个 Python 的 YAML 解析器。

CSS

cssutils – 一个 Python 的 CSS 库。

ATOM/RSS

feedparser – 通用的 feed 解析器。

SQL

sqlparse – 一个非验证的 SQL 语句分析器。

HTTP

HTTP

http-parser – C 语言实现的 HTTP 请求/响应消息解析器。

微格式

opengraph – 一个用来解析 Open Graph 协议标签的 Python 模块。

可移植的执行体

pefile – 一个多平台的用于解析和处理可移植执行体（即 PE）文件的模块。

PSD

psd-tools – 将 Adobe Photoshop PSD（即 PE）文件读取到 Python 数据结构。

自然语言处理

处理人类语言问题的库。

NLTK -编写 Python 程序来处理人类语言数据的最好平台。

Pattern – Python 的网络挖掘模块。他有自然语言处理工具，机器学习以及其它。

TextBlob – 为深入自然语言处理任务提供了一致的 API。是基于 NLTK 以及 Pattern 的巨人之肩上发展的。

jieba – 中文分词工具。

SnowNLP – 中文文本处理库。

loso – 另一个中文分词库。

genius – 基于条件随机域的中文分词。

langid.py – 独立的语言识别系统。

Korean – 一个韩文形态库。

pymorphy2 – 俄语形态分析器（词性标注+词形变化引擎）。

PyPLN – 用 Python 编写的分布式自然语言处理通道。这个项目的目标是创建一种简单的方法使用 NLTK 通过网络接口处理大语言库。

浏览器自动化与仿真

selenium – 自动化真正的浏览器（Chrome 浏览器，火狐浏览器，Opera 浏览器，IE 浏览器）。

Ghost.py – 对 PyQt 的 webkit 的封装（需要 PyQt）。

Spynner – 对 PyQt 的 webkit 的封装（需要 PyQt）。

Splinter – 通用 API 浏览器模拟器（selenium web 驱动，Django 客户端，Zope）。

多重处理

threading – Python 标准库的线程运行。对于 I/O 密集型任务很有效。对于 CPU 绑定的任务没用，因为 python GIL。

multiprocessing – 标准的 Python 库运行多进程。

celery – 基于分布式消息传递的异步任务队列/作业队列。

concurrent-futures – concurrent-futures 模块为调用异步执行提供了一个高层次的接口。

异步

异步网络编程库

asyncio – （在 Python 3.4 +版本以上的 Python 标准库）异步 I/O，时间循环，协同程序和任务。

Twisted – 基于事件驱动的网络引擎框架。

Tornado – 一个网络框架和异步网络库。

pulsar – Python 事件驱动的并发框架。

diesel – Python 的基于绿色事件的 I/O 框架。

gevent – 一个使用 greenlet 的基于协程的 Python 网络库。

eventlet – 有 WSGI 支持的异步框架。

Tomorrow – 异步代码的奇妙的修饰语法。

队列

celery – 基于分布式消息传递的异步任务队列/作业队列。

huey – 小型多线程任务队列。

mrq – Mr. Queue – 使用 redis & Gevent 的 Python 分布式工作任务队列。

RQ – 基于 Redis 的轻量级任务队列管理器。

simpleq – 一个简单的，可无限扩展，基于 Amazon SQS 的队列。

python-gearman – Gearman 的 Python API。

云计算

picloud – 云端执行 Python 代码。

dominoup.com – 云端执行 R，Python 和 matlab 代码。

电子邮件

电子邮件解析库

flanker – 电子邮件地址和 Mime 解析库。

Talon – Mailgun 库用于提取消息的报价和签名。

网址和网络地址操作

解析/修改网址和网络地址库。

URL

furl – 一个小的 Python 库，使得操纵 URL 简单化。

purl – 一个简单的不可改变的 URL 以及一个干净的用于调试和操作的 API。

urllib.parse – 用于打破统一资源定位器（URL）的字符串在组件（寻址方案，网络位置，路径等）之间的隔断，为了结合组件到一个 URL 字符串，并将“相对 URL”转化为一个绝对 URL，称之为“基本 URL”。

tlldextract – 从 URL 的注册域和子域中准确分离 TLD，使用公共后缀列表。

网络地址

netaddr – 用于显示和操纵网络地址的 Python 库。

网页内容提取

提取网页内容的库。

HTML 页面的文本和元数据

newspaper – 用 Python 进行新闻提取、文章提取和内容策展。

html2text – 将 HTML 转为 Markdown 格式文本。

python-goose – HTML 内容/文章提取器。

lassie – 人性化的网页内容检索工具

micawber – 一个从网址中提取丰富内容的小库。

sumy - 一个自动汇总文本文件和 HTML 网页的模块

Haul – 一个可扩展的图像爬虫。

python-readability – arc90 readability 工具的快速 Python 接口。

scrapely – 从 HTML 网页中提取结构化数据的库。给出了一些 Web 页面和数据提取的示例，scrapely 为所有类似的网页构建一个分析器。

视频

youtube-dl – 一个从 YouTube 下载视频的小命令程序。

you-get – Python3 的 YouTube、优酷/ Niconico 视频下载器。

维基

WikiTeam – 下载和保存 wikis 的工具。

WebSocket

用于 WebSocket 的库。

Crossbar – 开源的应用消息传递路由器（Python 实现的用于 Autobahn 的 WebSocket 和 WAMP）。

AutobahnPython – 提供了 WebSocket 协议和 WAMP 协议的 Python 实现并且开源。

WebSocket-for-Python – Python 2 和 3 以及 PyPy 的 WebSocket 客户端和服务端库。

DNS 解析

dnsyo – 在全球超过 1500 个的 DNS 服务器上检查你的 DNS。

pycares – c-ares 的接口。c-ares 是进行 DNS 请求和异步名称决议的 C 语言库。

计算机视觉

OpenCV – 开源计算机视觉库。

SimpleCV – 用于照相机、图像处理、特征提取、格式转换的简介，可读性强的接口（基于 OpenCV）。

mahotas – 快速计算机图像处理算法（完全使用 C++ 实现），完全基于 numpy 的数组作为它的数据类型。

代理服务器

shadowsocks – 一个快速隧道代理，可帮你穿透防火墙（支持 TCP 和 UDP，TFO，多用户和平滑重启，目的 IP 黑名单）。

tpoxy – tproxy 是一个简单的 TCP 路由代理（第 7 层），基于 Gevent，用 Python 进行配置。

其他 Python 工具列表

awesome-python

pycrumbs

python-github-projects

python_reference

pythonidae

4、 数据的来源（能够轻松的列举出 10 个网站）。

5、 数据的构成（抓取的某个字段在项目中有什么用）。

[\[python 爬虫的页面数据解析和提取/xpath/bs4/jsonpath/正则\(1\)\]](#)

一.数据类型及解析方式

一般来讲对我们而言，需要抓取的是某个网站或者某个应用的内容，提取有用的价值。内容一般分为两部分，非结构化的数据和结构化的数据。

- 非结构化数据：先有数据，再有结构，
- 结构化数据：先有结构、再有数据
- 不同类型的数据，我们需要采用不同的方式来处理。

1.非结构化的数据处理

文本、电话号码、邮箱地址

用:正则表达式

html 文件

用:正则表达式 / xpath/css 选择器/bs4

2.结构化的数据处理

json 文件

用:jsonPath / 转化成 Python 类型进行操作（json 类）

xml 文件

用:转化成 Python 类型（xmldict） / XPath / CSS 选择器 / 正则表达式

下面就将常用的数据解析及提取方式进行一下学习总结,主要包括:正则,bs4,jsonpath,xpath. json 数据优先选择使用 jsonpath. html 页面个人比较喜欢使用 xpath,若使用 xpath 较难提取的数据可以使用 bs4 进行辅助,若二者都提取不到,这时再去考虑使用正则,当然这只是个人建议,大神尽可全程高能使用正则.大佬们的牛逼人生,永远都不需要解释!!!!!!!!!!!!!!

=====邪恶的分割线=====

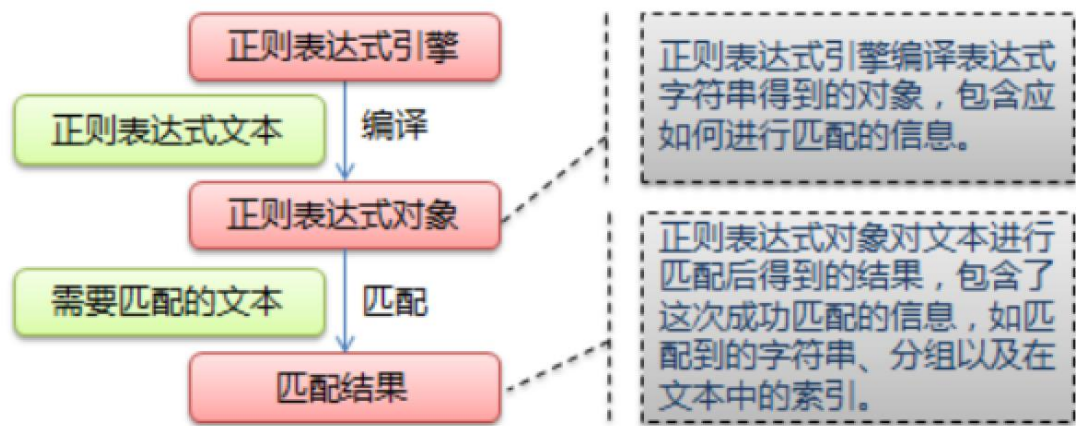
二.正则表达式

正则表达式，又称规则表达式，通常被用来检索、替换那些符合某个模式(规则)的文本。

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

给定一个正则表达式和另一个字符串，我们可以达到如下的目的：

- 给定的字符串是否符合正则表达式的过滤逻辑（“匹配”）；
- 通过正则表达式，从文本字符串中获取我们想要的特定部分（“过滤”）。



正则表达式匹配规则

语法	说明	表达式实例	完整匹配的字符串
字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade
预定义字符集（可以写在字符集[...]中）			
\d	数字：[0-9]	a\d c	a1c
\D	非数字：[^d]	a\D c	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\s c	a c
\S	非空白字符：[^s]	a\S c	abc
\w	单词字符：[A-Za-z0-9_]	a\w c	abc
\W	非单词字符：[^w]	a\W c	a c
数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	
边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
\$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc\$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^b]	a\Bbc	abc
逻辑、分组			
	代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果 没有被包括在()中，则它的范围是整个正则表达式。	abc def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的 仅在该组中有效。	(abc){2} a(123 456)c	abcaabc a456c
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcaabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5
特殊构造（不作为分组）			
(?:...)	(...)的不分组版本，用于使用' '或后接数量词。	(?:abc){2}	abcaabc
(?i msux)	i lmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(=?\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\d)a	前面是数字的a
(?<!=...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!=\d)a	前面不是数字的a
(?(id/name) yes-pattern no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。 no-pattern可以省略。	(\d)abc(?(1)\d abc)	1abc2 abcaabc

Python 的 re 模块的使用

在 Python 中，我们可以使用内置的 re 模块来使用正则表达式。

有一点需要特别注意的是，正则表达式使用对特殊字符进行转义，所以如果我们要使用原始字符串，只需加一个 r 前缀，示例：`r'chuanzhiboke\t\.\tpython'`

re 模块的一般使用步骤如下：

- b) 使用 `compile()` 函数将正则表达式的字符串形式编译为一个 `Pattern` 对象
- c) 通过 `Pattern` 对象提供的一系列方法对文本进行匹配查找，获得匹配结果，一个 `Match` 对象。
- d) 最后使用 `Match` 对象提供的属性和方法获得信息，根据需要进行其他的操作

compile 函数

`compile` 函数用于编译正则表达式，生成一个 `Pattern` 对象，它的一般使用形式如下：

```
import re
```

```
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')

```

在上面，我们已将一个正则表达式编译成 `Pattern` 对象，接下来，我们就可以利用 `pattern` 的一系列方法对文本进行匹配查找了。

`Pattern` 对象的一些常用方法主要有：

- `match` 方法：从起始位置开始查找，一次匹配
- `search` 方法：从任何位置开始查找，一次匹配
- `findall` 方法：全部匹配，返回列表
- `finditer` 方法：全部匹配，返回迭代器
- `split` 方法：分割字符串，返回列表
- `sub` 方法：替换

match 方法

`match` 方法用于查找字符串的头部（也可以指定起始位置），它是一次匹配，只要找到了一个匹配的结果(注意贪婪与非贪婪)就返回，而不是查找所有匹配的结果。它的一般使用形式如下：

```
>>> import re
>>> pattern = re.compile(r'\d+') # 用于匹配至少一个数字 + 一个或者无限多个

```

```
>>> m = pattern.match('one12twothree34four') # 查找头部，没有匹配
>>> print m
None

```

```
>>> m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配

```



```

>>> print m                                     # 从 0 开始算起
None

>>> m = pattern.match('one12twothree34four', 3, 10) # 从 '1' 的位置开始匹配，正好匹配
>>> print m                                     # 返回一个 Match 对象
<_sre.SRE_Match object at 0x10a42aac0>

>>> m.group(0)   # 可省略 0
'12'
>>> m.start(0)   # 可省略 0 起始位置
3
>>> m.end(0)     # 可省略 0 结束位置
5
>>> m.span(0)    # 可省略 0 区间
(3, 5)

```

再看看一个例子：

```

>>> import re
>>> pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I) # re.I 表示忽略大小写
>>> m = pattern.match('Hello World Wide Web')

>>> print m      # 匹配成功，返回一个 Match 对象
<_sre.SRE_Match object at 0x10bea83e8>

>>> m.group(0)   # 返回匹配成功的整个子串
'Hello World'

>>> m.span(0)    # 返回匹配成功的整个子串的索引
(0, 11)

>>> m.group(1)   # 返回第一个分组匹配成功的子串
'Hello'

>>> m.span(1)    # 返回第一个分组匹配成功的子串的索引
(0, 5)

>>> m.group(2)   # 返回第二个分组匹配成功的子串
'World'

>>> m.span(2)    # 返回第二个分组匹配成功的子串的索引
(6, 11)

>>> m.groups()  # 等价于 (m.group(1), m.group(2), ...)
('Hello', 'World')

>>> m.group(3)   # 不存在第三个分组
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group

```

search 方法

`search` 方法用于查找字符串的任何位置，它也是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果，当匹配成功时，返回一个 `Match` 对象，如果没有匹配上，则返回 `None`。让我们看看例子：

```
>>> import re
>>> pattern = re.compile('\d+')
>>> m = pattern.search('one12twothree34four') # 这里如果使用 match 方法则不匹配
>>> m
<_sre.SRE_Match object at 0x10cc03ac0>
>>> m.group()
'12'
>>> m = pattern.search('one12twothree34four', 10, 30) # 指定字符串区间
>>> m
<_sre.SRE_Match object at 0x10cc03b28>
>>> m.group()
'34'
>>> m.span()
(13, 15)
```

再来看一个例子：

```
# -*- coding: utf-8 -*-
```

```
import re
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
# 使用 search() 查找匹配的子串，不存在匹配的子串时将返回 None
# 这里使用 match() 无法成功匹配
m = pattern.search('hello 123456 789')
if m:
    # 使用 Match 获得分组信息
    print 'matching string:',m.group()
    # 起始位置和结束位置
    print 'position:',m.span()
```

执行结果：

```
matching string: 123456
position: (6, 12)
```

findall 方法

上面的 `match` 和 `search` 方法都是一次匹配，只要找到了一个匹配的结果就返回。然而，在大多数时候，我们需要搜索整个字符串，获得所有匹配的结果。

`findall` 以列表形式返回全部能匹配的子串，如果没有匹配，则返回一个空列表。

看看例子：

```
import re
pattern = re.compile(r'\d+') # 查找数字

result1 = pattern.findall('hello 123456 789')
result2 = pattern.findall('one1two2three3four4', 0, 10)
```

```
print result1
print result2
```

```
['123456', '789']
['1', '2']
```

再先看一个例子：

```
# re_test.py
```

```
import re
```

```
#re 模块提供一个方法叫 compile 模块，提供我们输入一个匹配的规则
```

```
#然后返回一个 pattern 实例，我们根据这个规则去匹配字符串
```

```
pattern = re.compile(r'\d+\.\d*')
```

```
#通过 partten.findall() 方法就能够全部匹配到我们得到的字符串
```

```
result = pattern.findall("123.141593, 'bigcat', 232312, 3.15")
```

```
#findall 以 列表形式 返回全部能匹配的子串给 result
```

```
for item in result:
```

```
    print item
```

```
123.141593
```

```
3.15
```

finditer 方法

`finditer` 方法的行为跟 `findall` 的行为类似，也是搜索整个字符串，获得所有匹配的结果。但它返回一个顺序访问每一个匹配结果（`Match` 对象）的迭代器。

```
# -*- coding: utf-8 -*-
```

```
import re
```

```
pattern = re.compile(r'\d+')
```

```
result_iter1 = pattern.finditer('hello 123456 789')
```

```
result_iter2 = pattern.finditer('one1two2three3four4', 0, 10)
```

```
print type(result_iter1)
```

```
print type(result_iter2)
```

```
print 'result1...'
```

```
for m1 in result_iter1:    # m1 是 Match 对象
```

```
    print 'matching string: {}, position: {}'.format(m1.group(), m1.span())
```

```
print 'result2...'
```

```
for m2 in result_iter2:
```

```
    print 'matching string: {}, position: {}'.format(m2.group(), m2.span())
```

执行结果：

```
<type 'callable-iterator'>
```

```
<type 'callable-iterator'>
```

```
result1...
```

```
matching string: 123456, position: (6, 12)
matching string: 789, position: (13, 16)
result2...
matching string: 1, position: (3, 4)
matching string: 2, position: (7, 8)
```

split 方法

split 方法按照能够匹配的子串将字符串分割后返回列表, 这个其实可算是变形的字符串方法 split() 但是使用正则的这个方法, 可以指定同时按照多个规则进行切割

```
split(string[, maxsplit])
```

其中, maxsplit 用于指定最大分割次数, 不指定将全部分割。

```
import re
p = re.compile(r'[\s\,\;]+')
print p.split('a,b;; c    d')
```

执行结果:

```
['a', 'b', 'c', 'd']
```

sub 方法

sub 方法用于替换。它的使用形式如下:

```
sub(repl, string[, count])
```

其中, repl 可以是字符串也可以是一个函数:

- 如果 repl 是字符串, 则会使用 repl 去替换字符串每一个匹配的子串, 并返回替换后的字符串, 另外, repl 还可以使用 id 的形式来引用分组, 但不能使用编号 0;
- 如果 repl 是函数, 这个方法应当只接受一个参数 (Match 对象), 并返回一个字符串用于替换 (返回的字符串中不能再引用分组)。
- count 用于指定最多替换次数, 不指定时全部替换。

```
import re
p = re.compile(r'(\w+) (\w+)') # \w = [A-Za-z0-9_]
s = 'hello 123, hello 456'
```

```
print p.sub(r'hello world', s) # 使用 'hello world' 替换 'hello 123' 和 'hello 456'
print p.sub(r'\2 \1', s)      # 引用分组
```

```
def func(m):
    return 'hi' + ' ' + m.group(2)
```

```
print p.sub(func, s)
print p.sub(func, s, 1)      # 最多替换一次
```

执行结果:

```
hello world, hello world
123 hello, 456 hello
```

```
hi 123, hi 456
hi 123, hello 456
```

=====邪恶的分割线=====

三.BS4-----CSS 选择器：BeautifulSoup4-----完美的汤

如果只是需要简单的使用 bs4 进行提取数据,可以拉到本小节最后面的红色字体开始的那部分!

一.官方文档：http://beautifulsoup.readthedocs.io/zh_CN/v4.4.0

二.Beautiful Soup 也是一个 HTML / XML 的解析器，主要的功能也是如何解析和提取 HTML / XML 数据。

lxml 只会局部遍历，而 BeautifulSoup 是基于 HTML DOM 的，会载入整个文档，解析整个 DOM 树，因此时间和**内存开销都会大很多，所以性能要低于lxml。**

BeautifulSoup 用来**解析HTML 比较简单，API 非常人性化**，支持 CSS 选择器，Python 标准库中的 HTML 解析器，也支持 lxml 的 XML 解析器。

Beautiful Soup 3 目前已经停止开发，推荐现在的项目使用 BeautifulSoup 4.使用 pip 安装即可：`pip install beautifulsoup4`

抓取工具	速度	使用难度	安装难度
正则	最快	困难	无（内置）
BeautifulSoup	慢	最简单	简单
LXML	快	简单	一般

三.简单的使用示例

```
from bs4 import BeautifulSoup

html = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""

#创建 BeautifulSoup 对象
soup = BeautifulSoup(html)

#打开本地 HTML 文件的方式来创建对象
#soup = BeautifulSoup(open('index.html'))

#格式化输出 soup 对象的内容
print soup.prettify()
```

输出结果:

```
<html>
<head>
  <title>
    The Dormouse's story
  </title>
</head>
<body>
  <p class="title" name="dromouse">
    <b>
      The Dormouse's story
    </b>
  </p>
  <p class="story">
    Once upon a time there were three little sisters; and their names were
    <a class="sister" href="http://example.com/elsie" id="link1">
      <!-- Elsie -->
    </a>
    ,
    <a class="sister" href="http://example.com/lacie" id="link2">
      Lacie
    </a>
    and
    <a class="sister" href="http://example.com/tillie" id="link3">
      Tillie
    </a>
    ;
    and they lived at the bottom of a well.
  </p>
  <p class="story">
    ...
  </p>
</body>
</html>
```

- 如果我们在 IPython2 下执行, 会看到这样一段警告:

```
/usr/local/lib/python2.7/site-packages/bs4/__init__.py:181: UserWarning: No parser
was explicitly specified, so I'm using the best available HTML parser for thi
s system ("lxml"). This usually isn't a problem, but if you run this code on ano
ther system, or in a different virtual environment, it may use a different parse
r and behave differently.
```

```
The code that caused this warning is on line 11 of the file /usr/local/bin/ipyth
on2. To get rid of this warning, change code that looks like this:
```

```
BeautifulSoup([your markup])
```

- 意思是, 如果我们没有显式地指定解析器, 所以默认使用这个系统的最佳可用 HTML 解析器 (“LXML”)。如果你在另一个系统中运行这段代码, 或者在不同的虚拟环境中, 使用不同的解析器造成行为不同。
- 但是可以通过 `soup = BeautifulSoup(html, “lxml”)` 方式指定 LXML 解析器。

四.bs4 的四大对象种类

Beautiful Soup 将复杂的 HTML 文档转换成一个复杂的树形结构，每个节点都是 Python 对象，所有对象可以归纳为 4 种：

- 标签
- NavigableString
- BeautifulSoup
- 评论

1. 标签

Tag 通俗点讲就是 HTML 中的一个标签，例如：

```
<head><title>The Dormouse's story</title></head>
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
```

上面的等等 title head a p HTML 标签加上里面包括的内容就是 Tag，那么试着使用 BeautifulSoup 来获取标签：

```
from bs4 import BeautifulSoup
```

```
html = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
```

```
#创建 BeautifulSoup 对象
soup = BeautifulSoup(html)
```

```
print soup.title
# <title>The Dormouse's story</title>
```

```
print soup.head
# <head><title>The Dormouse's story</title></head>
```

```
print soup.a
# <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
```

```
print soup.p
# <p class="title" name="dromouse"><b>The Dormouse's story</b></p>
```

```
print type(soup.p)
# <class 'bs4.element.Tag'>
```

我们可以利用汤加标签名轻松地获取这些标签的内容，但这些对象的类型是 `bs4.element.Tag`。但是注意，它查找的是在所有内容中的第一个符合要求的标签。如果要查询所有的标签，后面会进行介绍。

对于 *Tag*，它有两个重要的属性，是名和 *attrs*

```
print soup.name
# [document] #soup 对象本身比较特殊，它的 name 即为 [document]

print soup.head.name
# head #对于其他内部标签，输出的值便为标签本身的名称

print soup.p.attrs
# {'class': ['title'], 'name': 'dromouse'}
# 在这里，我们把 p 标签的所有属性打印输出了出来，得到的类型是一个字典。

print soup.p['class'] # soup.p.get('class')
# ['title'] #还可以利用 get 方法，传入属性的名称，二者是等价的

soup.p['class'] = "newClass"
print soup.p # 可以对这些属性和内容等等进行修改
# <p class="newClass" name="dromouse"><b>The Dormouse's story</b></p>

del soup.p['class'] # 还可以对这个属性进行删除
print soup.p
# <p name="dromouse"><b>The Dormouse's story</b></p>
```

2. NavigableString

既然我们已经得到了标签的内容，那么问题来了，我们要想获取标签内部的文字怎么办呢？很简单，用 *.string* 即可，例如

```
print soup.p.string
# The Dormouse's story

print type(soup.p.string)
# In [13]: <class 'bs4.element.NavigableString'>
```

3. BeautifulSoup

BeautifulSoup 对象表示的是一个文档的内容。大部分时候，可以把它当作 *Tag* 对象，是一个特殊的 *Tag*，我们可以分别获取它的类型，名称，以及属性来感受一下

```
print type(soup.name)
# <type 'unicode'>

print soup.name
# [document]

print soup.attrs # 文档本身的属性为空
# {}
```

4. 评论

注释对象是一个特殊类型的 *NavigableString* 对象，其输出的内容不包括注释符号。

```
print soup.a
# <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
```



```
print soup.a.string
# Elsie

print type(soup.a.string)
# <class 'bs4.element.Comment'>
```

a 标签里的内容实际上是注释，但是如果我们利用.string 来输出它的内容时，注释符号已经去掉了。

五.遍历文档树

1.直接子节点: .contents .children 属性

。内容

tag 的.content 属性可以将标签的子节点以列表的方式输出

```
print soup.head.contents
#[<title>The Dormouse's story</title>]
```

输出方式为列表，我们可以用列表索引来获取它的某一个元素

```
print soup.head.contents[0]
#<title>The Dormouse's story</title>
```

。孩子

它返回的不是一个列表，不过我们可以通过遍历获取所有子节点。

我们打印输出.children 看一下，可以发现它是一个 list 生成器对象

```
print soup.head.children
#<listiterator object at 0x7f71457f5710>
```

```
for child in soup.body.children:
    print child
```

结果:

```
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
```

```
<p class="story">Once upon a time there were three little sisters; and their names were
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
```

```
<p class="story">...</p>
```

2.所有子孙节点: .descendants 属性

.contents 和.children 属性仅包含标签的直接子节点，.descendants 属性可以对所有标签的子孙节点进行递归循环，和儿童类似，我们也需要遍历获取其中的内容。

```
for child in soup.descendants:
    print child
```

运行结果:

```
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
</body></html>
<head><title>The Dormouse's story</title></head>
<title>The Dormouse's story</title>
The Dormouse's story
```

```
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
</body>
```

```
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<b>The Dormouse's story</b>
The Dormouse's story
```

```
<p class="story">Once upon a time there were three little sisters; and their names were
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
Once upon a time there were three little sisters; and their names were
```

```
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
Elsie
,
```

```
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
Lacie
and
```

```
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
Tillie
;
and they lived at the bottom of a well.
```

```
<p class="story">...</p>
```

3.节点内容: .string 属性

如果一个标签里面没有标签了,那么.string 就会返回标签里面的内容。如果标签里面只有唯一的一个标签了,那么.string 也会返回最里面的内容。

```
print soup.head.string
#The Dormouse's story
print soup.title.string
#The Dormouse's story 六.搜索文档树
```

1.find_all(name, attrs, recursive, text, **kwargs)

1) 名称参数

name 参数可以查找所有名字为 name 的标签,字符串对象会被自动忽略掉

A. 传字符串

最简单的过滤器是字符串。在搜索方法中传入一个字符串参数,Beautiful Soup 会查找与字符串完整匹配的内容,下面的例子用于查找文档中所有的``标签:

```
soup.find_all('b')
# [<b>The Dormouse's story</b>]
```

```
print soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie" id="link1">!-- Elsie --></a>, <a c
lass="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" h
ref="http://example.com/tillie" id="link3">Tillie</a>]
```

B. 传正则表达式

如果传入正则表达式作为参数,Beautiful Soup 会通过正则表达式的 match () 来匹配内容。下面例子中找出所有以 b 开头的标签,这表示和标签都应该找到

```
import re
for tag in soup.find_all(re.compile("^b")):
    print(tag.name)
# body
# b
```

C. 传列表

如果传入列表参数,Beautiful Soup 会将与列表中任一元素匹配的内容返回。下面代码找到文档中所有标签和标签:

```
soup.find_all(["a", "b"])
# [<b>The Dormouse's story</b>,
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

2) 关键字参数

```
soup.find_all(class_ = "sister")
# [<a class="sister" href="http://example.com/elsie" id="link1">!-- Elsie --></a>, <a c
lass="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" h
ref="http://example.com/tillie" id="link3">Tillie</a>]
```

```
soup.find_all(id='link2')
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

3) 文字参数

通过文本参数可以搜索文档中的字符串内容，与名称参数的可选值一样，text 参数接受字符串，正则表达式，列表

```
soup.find_all(text="Elsie")
# [u'Elsie']

soup.find_all(text=["Tillie", "Elsie", "Lacie"])
# [u'Elsie', u'Lacie', u'Tillie']

soup.find_all(text=re.compile("Dormouse"))
[u"The Dormouse's story", u"The Dormouse's story"]
```

2.find()方法

find()的用法与 findall 一样，在于区别 find 用于查找第一个符合匹配查询查询结果，findall 则用于查找所有匹配查询查询结果的列表。

3. CSS 选择器(在爬虫中这是最常用的方式)

- 写 CSS 时，标签名不加任何修饰，类名前加英文句号 .，id 名前加 #
- 在这里我们也可以利用类似的方法来筛选元素，用到的方法是 soup.select()，返回类型是 list

(1) 通过标签名查找

```
from bs4 import BeautifulSoup
```

```
html = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
```

```
#创建 BeautifulSoup 对象
soup = BeautifulSoup(html)
```

```
print soup.select('title')
#[<title>The Dormouse's story</title>]
```

```
print soup.select('a') # 取到了所有的 a 标签
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a c
lass="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" h
ref="http://example.com/tillie" id="link3">Tillie</a>]
```

```
print soup.select('b')
#[<b>The Dormouse's story</b>]
```

（2）通过类名查找

```
print soup.select('.sister')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a c
lass="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" h
ref="http://example.com/tillie" id="link3">Tillie</a>]
```

（3）通过 id 名查找

```
print soup.select('#link1')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

（4）组合查找

组合查找即和写类文件时，标签名与类名，id 名进行的组合原理是一样的，例如查找 p 标签中，id 等于 link1 的内容，二者需要用空格分开

```
print soup.select('p #link1')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

直接子标签查找，使用则 > 分隔

```
print soup.select("head > title")
#[<title>The Dormouse's story</title>]
```

（5）属性查找

查找时还可以加入属性元素，属性需要用中括号括起来，注意属性和标签属于同一节点，所以中间不能加空格，否则会无法匹配到。

```
print soup.select('a[class="sister"]')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a c
lass="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" h
ref="http://example.com/tillie" id="link3">Tillie</a>]
```

```
print soup.select('a[href="http://example.com/elsie"]')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

同样，属性仍然可以与上述查找方式组合，不在同一节点的空格隔开，同一节点的不加空格

```
print soup.select('p a[href="http://example.com/elsie"]')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

（6）获取内容

```
soup = BeautifulSoup(html, 'lxml')
print type(soup.select('title'))
print soup.select('title')[0].get_text()
```

```
for title in soup.select('title'):
    print title.get_text()
```

bs4 基本实用的学习内容就是这些,更加详细完善的使用方法请查看官方文档

在 python 爬虫的页面数据解析和提取(2) 中 再继续记录爬虫数据解析余下的内容

链接: <https://www.cnblogs.com/lowmanisbusy/p/9226217.html>

6. 反反爬措施（限速、请求头、Cookie 池、代理池、Selenium、PhantomJS、RoboBrowser、TOR、OCR）。

一、UserAgent

UserAgent 中文名为用户代理，它使得服务器能够识别客户使用的操作系统及版本、CPU 类型、浏览器及版本等信息。对于一些网站来说，它会检查我们发送的请求中所携带的 UserAgent 字段，如果非浏览器，就会被识别为爬虫，一旦被识别出来，我们的爬虫也就无法正常爬取数据了。这里先看一下在不设置 UserAgent 字段时该字段的值会是什么

2.使用第三方库--fake_useragent

使用方法如下：

```
1 from fake_useragent import UserAgent
2
3
4 ua = UserAgent()
5 for i in range(3):
6     print(ua.random)
7 # Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.145
3.93 Safari/537.36
8 # Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.222
8.0 Safari/537.36
9 # Mozilla/5.0 (X11; Linux i686; rv:21.0) Gecko/20100101 Firefox/21.0
```

二、IP

对于一些网站来说，如果某个 IP 在单位时间里的访问次数超过了某个阈值，那么服务器就会 ban 掉这个 IP 了，它就会返回给你一些错误的数据。一般来说，当我们的 IP 被 ban 了，我们的爬虫也就无法正常获取数据了，但是用浏览器还是可以正常访问，但是如果用浏览器都无法访问，那就真的 GG 了。很多网站都会对 IP 进行检测，比如知乎，如果单个 IP 访问频率过高就会被封掉。

解决办法：

使用代理 IP。网上有很多免费代理和付费代理可供选择，免费代理比如：[西刺代理](#)、[快代理](#)等等，付费代理比如：[代理云](#)、[阿布云](#)等等。除此之外，我们还可以建一个属于自己的代理池以供使用，这里可以参考下我的[上一篇博客](#)。

三（01）、Referer 防盗链

防盗链主要是针对客户端请求过程中所携带的一些关键信息来验证请求的合法性，而防盗链又有很多种，比如 Referer 防盗链、时间戳防盗链等等，这里只讲 Referer 防盗链。Referer 用于告知服务器该请求是从哪个页面链接过来的，比如我们先打开[少司命](#)的百度百科：解决办法：在请求头 headers 中添加 Referer 字段以及相应的值。

三（02）、防盗链

这次我遇到的防盗链，除了前面说的 Referer 防盗链，还有 Cookie 防盗链和时间戳防盗链。Cookie 防盗链常见于论坛、社区。当访客请求一个资源的时候，他会检查这个访客的 Cookie，如果不是他自己的用户的 Cookie，就不会给这个访客正确的资源，也就达到了防盗的目的。时间戳防盗链指的是在他的 url 后面加上一个时间戳参数，所以如果你直接请求网站的 url 是无法得到真实的页面的，只有带上时间戳才可以。

四、在 html 中动手脚

首先我不得不佩服那些前端工程师们，为了反爬虫真是想了不少办法，比如 JS 加密啊 JS 混淆啊，真是搞得人头大。不过我们这里先说那些在 html 中动手脚的，比如加一些无意义的字符之类的，这样即使我们能爬下来，得到的数据也是没法使用的。比如部分微信公众号的文章里会穿插一些乱七八糟的字符，这里用[这篇文章](#)作为例子：解决办法：可以看到每个字前面都加了一个 span 标签，span 标签里加入了一个用于干扰的字符，而且有的还使用了 strong 标签，这就给我们的解析增加了难度。这里我使用的是 lxml 解析，解析完之后再对数据做一下清洗，完整代码如下略：

五.随机化网页源码

用 display:none 来随机化网页源码，有网站还会随机类和 id 的名字，然后再加点随机的 tr 和 td，这样的话就增大了我们解析的难度。比如[全网代理 IP](#)：解决办法：可以看到每个 IP 都是包含在一个 class 为“ip”的 td 里的，所以我们可以先定位到这个 td，然后进行下一步解析。虽然这个 td 里面包含了很多的 span 标签和 p 标签，而且也每个标签的位置也没有什么规律，不过还是有办法解析的。方法就是把这个 td 里的所有文字提取出来，然后把那些前后重复的部分去除掉，最后拼接到一起就可以了，代码如下略：最后就能得到我们想要的數據了。不过我们得到的端口数据和网页上显示的数据是不一样的，这是因为端口数据是经过 JS 混淆的，至于怎么破解，下次会分享出来。

六、全网代理 IP 的 JS 混淆

首先进入[全网代理 IP](#)，打开开发者工具，点击查看端口号，看起来貌似没有什么问题：

七、用图片代替文字

之前就有人评论说有的网站使用图片代替文字以实现反爬虫，然后我这次就找到了一个网站--[新蛋网](#)，随意点击一个商品查看一下：解决办法：我找到两个可以得到价格的办法，一个简单的，一个难一点的。简单的方法是用正则表达式，因为在源码中的其他地方是包含商品的基本信息的，比如名称和价格，所以我们可以使用正则表达式进行匹配，难一点的方法是把图片下载到本地之后进行识别，由于这个图片的清晰度很高，也没有扭曲或者加入干扰线什么的，所以可以直接使用 OCR 进行识别。但是用这种方法的话需要安装好 Tesseract-OCR，这个工具的安装过程还是比较麻烦的。

0x01 常见的反爬虫

这几天在爬一个网站，网站做了很多反爬虫工作，爬起来有些艰难，花了一些时间才绕过反爬虫。在这里把我写爬虫以来遇到的各种反爬虫策略和应对的方法总结一下。

从功能上来讲，爬虫一般分为数据采集，处理，储存三个部分。这里我们只讨论数据采集部分。

一般网站从三个方面反爬虫：用户请求的 Headers，用户行为，网站目录和数据加载方式。前两种比较容易遇到，大多数网站都从这些角度来反爬虫。第三种一些应用 ajax 的网站会采用，这样增大了爬取的难度。

0x02 通过 Headers 反爬虫

从用户请求的 Headers 反爬虫是最常见的反爬虫策略。很多网站都会对 Headers 的 User-Agent 进行检测，还有一部分网站会对 Referer 进行检测（一些资源网站的防盗链就是检测 Referer）。如果遇到了这类反爬虫机制，可以直接在爬虫中添加 Headers，将浏览器的 User-Agent 复制到爬虫的 Headers 中；或者将 Referer 值修改为目标网站域名。对于检测 Headers 的反爬虫，在爬虫中修改或者添加 Headers 就能很好的绕过。

0x03 基于用户行为反爬虫

还有一部分网站是通过检测用户行为，例如同一个 IP 短时间内多次访问同一页面，或者同一账户短时间内多次进行相同操作。

大多数网站都是前一种情况，对于这种情况，使用 IP 代理就可以解决。可以专门写一个爬虫，爬取网上公开的代理 ip，检测后全部保存起来。这样的代理 ip 爬虫经常会用到，最好自己准备一个。有

了大量代理 ip 后可以每请求几次更换一个 ip，这在 requests 或者 urllib2 中很容易做到，这样就能很容易的绕过第一种反爬虫。

对于第二种情况，可以在每次请求后随机间隔几秒再进行下一次请求。有些有逻辑漏洞的网站，可以通过请求几次，退出登录，重新登录，继续请求来绕过同一账号短时间内不能多次进行相同请求的限制。

0x04 动态页面的反爬虫

上述的几种情况大多都是出现在静态页面，还有一部分网站，我们需要爬取的数据是通过 ajax 请求得到，或者通过 JavaScript 生成的。首先用 Firebug 或者 HttpFox 对网络请求进行分析。如果能够找到 ajax 请求，也能分析出具体的参数和响应的具体含义，我们就能采用上面的方法，直接利用 requests 或者 urllib2 模拟 ajax 请求，对响应的 json 进行分析得到需要的数据。

能够直接模拟 ajax 请求获取数据固然是极好的，但是有些网站把 ajax 请求的所有参数全部加密了。我们根本没办法构造自己所需要的数据的请求。我这几天爬的那个网站就是这样，除了加密 ajax 参数，它还把一些基本的功能都封装了，全部都是在调用自己的接口，而接口参数都是加密的。遇到这样的网站，我们就不能用上面的方法了，我用的是 selenium+phantomJS 框架，调用浏览器内核，并利用 phantomJS 执行 js 来模拟人为操作以及触发页面中的 js 脚本。从填写表单到点击按钮再到滚动页面，全部都可以模拟，不考虑具体的请求和响应过程，只是完完整整的把人浏览页面获取数据的过程模拟一遍。

用这套框架几乎能绕过大多数的反爬虫，因为它不是在伪装成浏览器来获取数据（上述的通过添加 Headers 一定程度上就是为了伪装成浏览器），它本身就是浏览器，phantomJS 就是一个没有界面的浏览器，只是操控这个浏览器的不是人。利用 selenium+phantomJS 能干很多事情，例如识别点触式（12306）或者滑动式的验证码，对页面表单进行暴力破解等等。它在自动化渗透中还会大展身手，以后还会提到这个。

八.设置合理的 cookie

虽然 cookie 是一把双刃剑，但正确地处理 cookie 可以避免许多采集问题。网站会用 cookie 跟踪你的访问过程，如果发现了爬虫异常行为就会中断你的访问，比如特别快速地填写表单，或者浏览大量页面。虽然这些行为可以通过关闭并重新连接或者改变 IP 地址来伪装，但是如果 cookie 暴露了你的身份，再多努力也是白费。

在采集一些网站时 cookie 是不可或缺的。要在一个网站上持续保持登录状态，需要在多个页面中保存一个 cookie。有些网站不要求在每次登录时都获得一个新 cookie，只要保存一个旧的“已登录”的 cookie 就可以访问。

如果你在采集一个或者几个目标网站，建议你检查这些网站生成的 cookie，然后想想哪一个 cookie 是爬虫需要处理的。有一些浏览器插件可以为你显示访问网站和离开网站时 cookie 是如何设置的。

EditThisCookie (<http://www.editthiscookie.com/>) 是我最喜欢的 Chrome 浏览器插件之一。

因为 requests 模块不能执行 JavaScript，所以它不能处理很多新式的跟踪软件生成的 cookie，比如 Google Analytics，只有当客户端脚本执行后才设置 cookie（或者在用户浏览页面时基于网页事件产生 cookie，比如点击按钮）。要处理这些动作，需要用 Selenium 和 PhantomJS 包。

Selenium 与 PhantomJS

Selenium (<http://www.seleniumhq.org/>) 是一个强大的网络数据采集工具，最初是为网站自动化测试而开发的。近几年，它还被广泛用于获取精确的网站快照，因为它们可以直接运行在浏览器上。

Selenium 可以让浏览器自动加载页面，获取需要的数据，甚至页面截屏，或者判断网站上某些动作是否发生。

Selenium 自己不带浏览器，它需要与第三方浏览器结合在一起使用。例如，如果你在 **Firefox** 上运行 **Selenium**，可以直接看到 **Firefox** 窗口被打开，进入网站，然后执行你在代码中设置的动作。虽然这样可以看得更清楚，但是我更喜欢让程序在后台运行，所以我 **PhantomJS**

（<http://phantomjs.org/download.html>）代替真实的浏览器。

PhantomJS 是一个“无头”（**headless**）浏览器。它会把网站加载到内存并执行页面上的 **JavaScript**，但不会向用户展示网页的图形界面。将 **Selenium** 和 **PhantomJS** 结合在一起，就可以运行一个非常强大的网络爬虫了，可以处理 **cookie**、**JavaScript**、**header**，以及任何你需要做的事情。

可以从 **PyPI** 网站（<https://pypi.python.org/simple/selenium/>）下载 **Selenium** 库，也可以用第三方管理器（像 **pip**）用命令行安装。

你可以对任意网站（本例用的是 <http://pythonscraping.com>）调用 **webdriver** 的 **get_cookie()** 方法来查看 **cookie**：

这样就可以获得一个非常典型的 **Google Analytics** 的 **cookie** 列表：

还可以调用 **deletetecookie()**、**addcookie()** 和 **deleteallcookies()** 方法来处理 **cookie**。另外，还可以保存 **cookie** 以备其他网络爬虫使用。下面的例子演示了如何把这些函数组合在一起：

在这个例子中，第一个 **webdriver** 获得了一个网站，打印 **cookie** 并把它们保存到变量 **savedCookies** 里。第二个 **webdriver** 加载同一个网站（技术提示：必须首先加载网站，这样 **Selenium** 才能知道 **cookie** 属于哪个网站，即使加载网站的行为对我们没任何用处），删除所有的 **cookie**，然后替换成第一个 **webdriver** 得到的 **cookie**。当再次加载这个页面时，两组 **cookie** 的时间戳、源代码和其他信息应该完全一致。从 **Google Analytics** 的角度看，第二个 **webdriver** 现在和第一个 **webdriver** 完全一样。

3. 爬虫访问速度和路径的控制

有一些防护措施完备的网站可能会阻止你快速地提交表单，或者快速地与网站进行交互。即使没有这些安全措施，用一个比普通人快很多的速度从一个网站下载大量信息也可能让自己被网站封杀。

因此，虽然多线程程序可能是一个快速加载页面的好办法——在一个线程中处理数据，另一个线程中加载页面——但是这对编写好的爬虫来说是恐怖的策略。还是应该尽量保证一次加载页面加载且数据请求最小化。如果条件允许，尽量为每个页面访问增加一点儿时间间隔，即使你要增加一行代码：

```
time.sleep(3)
```

（小编：3 + 随机数 是不是更好一些？）

合理控制速度是你不应该破坏的规则。过度消耗别人的服务器资源会让你置身于非法境地，更严重的是这么做可能会把一个小型网站拖垮甚至下线。拖垮网站是不道德的，是彻头彻尾的错误。所以请控制采集速度！

常见表单反爬虫安全措施解密

许多像 **Litmus** 之类的测试工具已经用了很多年了，现在仍用于区分网络爬虫和使用浏览器的人类访问者，这类手段都取得了不同程度的效果。虽然网络机器人下载一些公开的文章和博文并不是什么大事，但是如果网络机器人在你的网站上创造了几千个账号并开始向所有用户发送垃圾邮件，就是一个大问题了。网络表单，尤其是那些用于账号创建和登录的网站，如果被机器人肆意地滥用，网站的安

全和流量费用就会面临严重威胁，因此努力限制网站的接入是最符合许多网站所有者的利益的（至少他们这么认为）。

这些集中在表单和登录环节上的反机器人安全措施，对网络爬虫来说确实是严重的挑战。

4. 注意网页隐藏的信息

在 HTML 表单中，“隐含”字段可以让字段的值对浏览器可见，但是对用户不可见（除非看网页源代码）。随着越来越多的网站开始用 `cookie` 存储状态变量来管理用户状态，在找到另一个最佳用途之前，隐含字段主要用于阻止爬虫自动提交表单。

下图显示的例子就是 Facebook 登录页面上的隐含字段。虽然表单里只有三个可见字段（`username`、`password` 和一个确认按钮），但是在源代码里表单会向服务器传送大量的信息。

Facebook 登录页面上的隐含字段

用隐含字段阻止网络数据采集的方式主要有两种。第一种是表单页面上的一个字段可以用服务器生成的随机变量表示。如果提交时这个值不在表单处理页面上，服务器就有理由认为这个提交不是从原始表单页面上提交的，而是由一个网络机器人直接提交到表单处理页面的。绕开这个问题的最佳方法就是，首先采集表单所在页面上生成的随机变量，然后再提交到表单处理页面。

第二种方式是“蜜罐”（`honey pot`）。如果表单里包含一个具有普通名称的隐含字段（设置蜜罐圈套），比如“用户名”（`username`）或“邮箱地址”（`email address`），设计不太好的网络机器人往往不管这个字段是不是对用户可见，直接填写这个字段并向服务器提交，这样就会中服务器的蜜罐圈套。服务器会把所有隐含字段的真实值（或者与表单提交页面的默认值不同的值）都忽略，而且填写隐含字段的访问用户也可能被网站封杀。

总之，有时检查表单所在的页面十分必要，看看有没有遗漏或弄错一些服务器预先设定好的隐含字段（蜜罐圈套）。如果你看到一些隐含字段，通常带有较大的随机字符串变量，那么很可能网络服务器会在表单提交的时候检查它们。另外，还有其他一些检查，用来保证这些当前生成的表单变量只被使用一次或是最近生成的（这样可以避免变量被简单地存储到一个程序中反复使用）。

5. 避免进入蜜罐

虽然在进行网络数据采集时用 CSS 属性区分有用信息和无用信息会很容易（比如，通过读取 `id` 和 `class` 标签获取信息），但这么做有时也会出问题。如果网络表单的一个字段通过 CSS 设置成对用户不可见，那么可以认为普通用户访问网站的时候不能填写这个字段，因为它没有显示在浏览器上。如果这个字段被填写了，就可能是机器人干的，因此这个提交会失效。

这种手段不仅可以应用在网站的表单上，还可以应用在链接、图片、文件，以及一些可以被机器人读取，但普通用户在浏览器上却看不到的任何内容上面。访问者如果访问了网站上的一个“隐含”内容，就会触发服务器脚本封杀这个用户的 IP 地址，把这个用户踢出网站，或者采取其他措施禁止这个用户接入网站。实际上，许多商业模式就是在干这些事情。

下面的例子所用的网页在 <http://pythonscraping.com/pages/itsatrap.html>。这个页面包含了两个链接，一个通过 CSS 隐含了，另一个是可见的。另外，页面上还包括两个隐含字段：

这三个元素通过三种不同的方式对用户隐藏：

- 第一个链接是通过简单的 CSS 属性设置 `display:none` 进行隐藏
- 电话号码字段 `name="phone"` 是一个隐含的输入字段

- 邮箱地址字段 `name="email"` 是将元素向右移动 50 000 像素（应该会超出电脑显示器的边界）并隐藏滚动条

因为 Selenium 可以获取访问页面的内容，所以它可以区分页面上的可见元素与隐含元素。通过 `is_displayed()` 可以判断元素在页面上是否可见。

例如，下面的代码示例就是获取前面那个页面的内容，然后查找隐含链接和隐含输入字段：

Selenium 抓取出了每个隐含的链接和字段，结果如下所示：

虽然你不太可能会去访问你找到的那些隐含链接，但是在提交前，记得确认一下那些已经在表单中、准备提交的隐含字段的值（或者让 Selenium 为你自动提交）。

6. 使用可变 IP

建立网络爬虫的第一原则是：所有信息都可以伪造。你可以用非本人的邮箱发送邮件，通过命令行自动化鼠标的行为，或者通过 IE 5.0 浏览器耗费网站流量来吓唬网管。

但是有一件事情是不能作假的，那就是你的 IP 地址。任何人都可以用这个地址给你写信：“美国华盛顿特区宾夕法尼亚大道西北 1600 号，总统，邮编 20500。”但是，如果这封信是从新墨西哥州的阿尔伯克基市发来的，那么你肯定可以确信给你写信的不是美国总统。

从技术上说，IP 地址是可以通过发送数据包进行伪装的，就是分布式拒绝服务攻击技术（Distributed Denial of Service, DDoS），攻击者不需要关心接收的数据包（这样发送请求的时候就可以使用假 IP 地址）。但是网络数据采集是一种需要关心服务器响应的行为，所以我们认为 IP 地址是不能造假的。

阻止网站被采集的注意力主要集中在识别人类与机器人的行为差异上面。封杀 IP 地址这种矫枉过正的行为，就好像是农民不靠喷农药给庄稼杀虫，而是直接用火烧彻底解决问题。它是最后一步棋，不过是一种非常有效的方法，只要忽略危险 IP 地址发来的数据包就可以了。但是，使用这种方法会遇到以下几个问题。

- IP 地址访问列表很难维护。虽然大多数大型网站都会用自己的程序自动管理 IP 地址访问列表（机器人封杀机器人），但是至少需要人偶尔检查一下列表，或者至少要监控问题的增长。
- 因为服务器需要根据 IP 地址访问列表去检查每个准备接收的数据包，所以检查接收数据包时会额外增加一些处理时间。多个 IP 地址乘以海量的数据包更会使检查时间指数级增长。为了降低处理时间和处理复杂度，管理员通常会对 IP 地址进行分组管理并制定相应的规则，比如如果这组 IP 中有一些危险分子就“把这个区间的所有 256 个地址全部封杀”。于是产生了下一个问题。
- 封杀 IP 地址可能会导致意外后果。例如，当我还在美国麻省欧林工程学院读本科的时候，有个同学写了一个可以在 <http://digg.com/> 网站（在 Reddit 流行之前大家都用 Digg）上对热门内容进行投票的软件。这个软件的服务器 IP 地址被 Digg 封杀，导致整个网站都不能访问。于是这个同学就把软件移到了另一个服务器上，而 Digg 自己却失去了许多主要目标用户的访问量。

虽然有这些缺点，但封杀 IP 地址依然是一种十分常用的手段，服务器管理员用它来阻止可疑的网络爬虫入侵服务器。

Tor 代理服务器

洋葱路由（The Onion Router）网络，常用缩写为 Tor，是一种 IP 地址匿名手段。由网络志愿者服务器构建的洋葱路由器网络，通过不同服务器构成多个层（就像洋葱）把客户端包在最里面。数据进入网络之前会被加密，因此任何服务器都不能偷取通信数据。另外，虽然每一个服务器的入站和出站通

信都可以被查到，但是要想查出通信的真正起点和终点，必须知道整个通信链路上所有服务器的入站和出站通信细节，而这基本是不可能实现的。

Tor 匿名的局限性

虽然我们在本文中用 Tor 的目的是改变 IP 地址，而不是实现完全匿名，但有必要关注一下 Tor 匿名方法的能力和不足。

虽然 Tor 网络可以让你访问网站时显示的 IP 地址是一个不能跟踪到你的 IP 地址，但是你在网站上留给服务器的任何信息都会暴露你的身份。例如，你登录 Gmail 账号后再用 Google 搜索，那些搜索历史就会和你的身份绑定在一起。

另外，登录 Tor 的行为也可能让你的匿名状态处于危险之中。2013 年 12 月，一个哈佛大学本科生想逃避期末考试，就用一个匿名邮箱账号通过 Tor 网络给学校发了一封炸弹威胁信。结果哈佛大学的 IT 部门通过日志查到，在炸弹威胁信发来的时候，Tor 网络的流量只来自一台机器，而且是一个在校学生注册的。虽然他们不能确定流量的最初源头（只知道是通过 Tor 发送的），但是作案时间和注册信息证据充分，而且那个时间段内只有一台机器是登录状态，这就有充分理由起诉那个学生了。

登录 Tor 网络不是一个自动的匿名措施，也不能让你进入互联网上任何区域。虽然它是一个实用的工具，但是用它的时候一定要谨慎、清醒，并且遵守道德规范。

在 Python 里使用 Tor，需要先安装运行 Tor，下一节将介绍。Tor 服务很容易安装和开启。只要去 Tor 下载页面下载并安装，打开后连接就可以。不过要注意，当你用 Tor 的时候网速会变慢。这是因为代理有可能要先在全世界网络上转几次才到目的地！

PySocks

PySocks 是一个非常简单的 Python 代理服务器通信模块，它可以和 Tor 配合使用。你可以从它的网站（<https://pypi.python.org/pypi/PySocks>）上下载，或者使用任何第三方模块管理器安装。

这个模块的用法很简单。示例代码如下所示。运行的时候，Tor 服务必须运行在 9150 端口（默认值）上：

网站 <http://icanhazip.com/> 会显示客户端连接的网站服务器的 IP 地址，可以用来测试 Tor 是否正常运行。当程序执行之后，显示的 IP 地址就不是你原来的 IP 了。

如果你想在 Tor 里面用 Selenium 和 PhantomJS，不需要 PySocks，只要保证 Tor 在运行，然后增加 service_args 参数设置代理端口，让 Selenium 通过端口 9150 连接网站就可以了：

和之前一样，这个程序打印的 IP 地址也不是你原来的，而是你通过 Tor 客户端获得的 IP 地址。

从网站主机运行

如果你拥有个人网站或公司网站，那么你可能已经知道如何使用外部服务器运行你的网络爬虫了。即使是一些相对封闭的网络服务器，没有可用的命令行接入方式，你也可以通过网页界面对程序进行控制。

如果你的网站部署在 Linux 服务器上，应该已经运行了 Python。如果你用的是 Windows 服务器，可能就没那么幸运了；你需要仔细检查一下 Python 有没有安装，或者问问网管可不可以安装。

大多数小型网络主机都会提供一个软件叫 cPanel，提供网站管理和后台服务的基本管理功能和信息。如果你接入了 cPanel，就可以设置 Python 在服务器上运行——进入“Apache Handlers”然后增加一个 handler（如还没有的话）：

这会告诉服务器所有的 Python 脚本都将作为一个 CGI 脚本运行。CGI 就是通用网关接口（Common Gateway Interface），是可以在服务器上运行的任何程序，会动态地生成内容并显示在网站上。把 Python 脚本显式地定义成 CGI 脚本，就是给服务器权限去执行 Python 脚本，而不只是在浏览器上显示它们或者让用户下载它们。

写完 Python 脚本后上传到服务器，然后把文件权限设置成 755，让它可执行。通过浏览器找到程序上传的位置（也可以写一个爬虫来自动做这件事情）就可以执行程序。如果你担心在公共领域执行脚本不安全，可以采取以下两种方法。

- 把脚本存储在一个隐晦或深层的 URL 里，确保其他 URL 链接都不能接入这个脚本，这样可以避免搜索引擎发现它。
- 用密码保护脚本，或者在执行脚本之前用密码或加密令牌进行确认。

确实，通过这些原本主要是用来显示网站的服务运行 Python 脚本有点儿复杂。比如，你可能会发现网络爬虫运行时网站的加载速度变慢了。其实，在整个采集任务完成之前页面都是不会加载的（得等到所有“print”语句的输出内容都显示完）。这可能会消耗几分钟，几小时，甚至永远也完成不了，要看程序的具体情况了。虽然它最终一定能完成任务，但是可能你还想看到实时的结果，这样就需要一台真正的服务器了。

从云主机运行

虽然云计算的花费可能是无底洞，但是写这篇文章时，启动一个计算实例最便宜只要每小时 1.3 美分（亚马逊 EC2 的 micro 实例，其他实例会更贵），Google 最便宜的计算实例是每小时 4.5 美分，最少需要用 10 分钟。考虑计算能力的规模效应，从大公司买一个小型的云计算实例的费用，和自己买一台专业实体机的费用应该差不多——不过用云计算不需要雇人去维护设备。

设置好计算实例之后，你就有了新 IP 地址、用户名，以及可以通过 SSH 进行实例连接的公私密钥了。后面要做的每件事情，都应该和你在实体服务器上干的事情一样了——当然，你不需要再担心硬件维护，也不用运行复杂多余的监控工具了。

转载自：<http://www.cnblogs.com/junrong624/p/5533655.html>

6、数据的体量（最后抓取了多少数据，多少 W 条数据或多少个 G 的数据）。

爬虫爬下来的数据(100G 级别，2000W 以上数据量)用 mysql 还是 mongodb 存储好？

MongoDB 作为非关系型数据库，其主要的优势在于 schema-less。由于爬虫数据一般来说比较“脏”，不会包含爬取数据的所有 field，这对于不需要严格定义 schema 的 MongoDB 再合适不过。而 MongoDB 内置的 sharding 分布式系统也保证了它的可扩展性。MongoDB 的 aggregation framework 除了 join 以外可以完全替代 SQL 语句，做到非常快速的统计分析。

而题主的 100GB、20m 数据量(5k per record)，据我的经验，这对于 MongoDB 来说不是太大问题，需要全局统计的话就做 sharding+自带的 Map Reduce 进行优化，需要 filter 的话就做索引（前人也提到 MongoDB 的查询速度是 MySQL 不能比的），而且需要 join 的概率也不大（不需要 normalize）。（推荐相关 mysql 视频教程：[mysql 教程](#)）

总而言之，主要看你用来做什么，如果是简单的 raw data 储存直接存成 txt 文件，后续加载到 HDFS 都可以。如果是数据仓库设计的话，MySQL 可以作为一个轻量级的 aggregate table 载体，作为 OLAP 的后端数据源。（推荐相关 MongoDB 视频教程：[MongoDB 教程](#)）

反正，在这种情况下，我是看不到 MySQL 单纯用做储存的必要。well，这个量级，且用处来看，mysql or mongo 都无所谓，区别不大。不过你既然爬虫的数据，就会要跟着源数据结构变动而变动，mongo 的模式就会更方便适合些

Mongo 快的原因主要有以下几点：

写：-3.0 以前是 mmap，内存映射模式，写入数据时候在内存里完成后就是可以返回的，这样并发会高，当然也有各种级别的安全写级别，应对不同的安全需求。

-3.0 之后，WT 引擎其 MVCC 机制更是大幅度提高了写效率，多版本控制机制提高了并发，降低了锁的粒度。

读：MongoDB 与 mysql 不同，文档型的结构设计使同一条 document 中的内容在连续的位置内（内存啊，硬盘啊）。而关系型数据库需要把数据从各个地方找过来，join 啊之类的，减少了随机 io。

Mongo 的设计模式也会让我们尽可能的把 working set 能在 ram 中装下。

3.0 以后 WT 的 MVCC 也大幅度提高了效率。

然后，sharding 的存在，让我们对于带有 shard key 的读与写都有了横向水平扩展的能力，也提高了效率。这个问题是钓鱼吧...

存 100G，2KW 条数据随便一个数据库都妥妥的，真正用来做选择的是要怎么用... 那个熟悉用那个。我觉得要看你的爬虫是用什么语言吧，php 的就 mysql 比较好点，nodejs 就 mongodb 呗～还有就是数据结构要考虑考虑，还有读写要求你也没给出这些很难推荐，不过我推崇芒果，因为跟 node 无缝对接，不好就是比较新，坑也多～这样以后数据有了直接来 MEAN 框架就可以弄爬虫应用了 O(n_n)O

首先，数据大了，存储绝对不是一件容易的事，要考虑很多因素。

爬虫爬下来的大量数据，存在关系型数据库里往往不是很恰当的，因为当数据量和并发很大时，关系型数据库的容量与读写能力会是瓶颈，另一方面，爬虫保存的页面信息之间一般也不需要建立关系。

比较好的做法应该是存在列族数据库类型的 Nosql 里。Google 的 BigTable 论文里就提出了使用 BigTable 存储网页信息，开源的列族数据库，像 HBase、Cassandra 也都很适合存储这类信息。每爬一个网页，构造一个 Key（比如是倒排域名的 url，或者是散列的 key）和一系列 Column（网页内容等），插到 HBase 的里，作为一行。

有一套较通用的大规模分布式爬虫方案是 Nutch + Gora + HBase + Solr / Elasticsearch，爬虫爬的数据通过 Gora 作为数据抽象层存在 HBase 里，然后导入 Solr 或者 Elasticsearch 里建立索引。也可以通过 Gora 执行 MapReduce 或者导入 Spark 进行计算。

但是上述方案其实并不适合普通的开发者，因为搭建和维护 HBase 是很繁琐的，引入很多学习成本，遇到问题还要排查。重要的是这跟爬虫毫无关系啊，完全是存储问题。

所以我最终推荐的是云的方案，阿里云的 OTS 是一个类似 HBase 的 Nosql 数据库，成本低、读写性能好，非常适合爬虫这个场景：

- a) 不需要自己搭建与运维，开通实例即可使用，完全不用担心规模问题。
- b) 按照读写预留能力收费，爬虫爬的时候读写预留能力调上去，爬完了读写预留能力调下来。
- c) 存储成本非常低。
- d) 数据存在 OTS 上，计算资源就可以弹性的扩容或者缩减。举个例子，假如爬虫爬的时候要使用很多云服务器，等爬虫爬完了，这些服务器就可以及时释放；另一方面，如果要对爬下来的数据做分析计算，也只需要在计算的时候购买云服务器，从 OTS 中把数据导下来，计算完成服务器即可释放。

开放结构化数据服务 OTS_海量数据存储

利益相关：OTS 开发

都是些什么数据？拿到后做什么处理？query pattern 是啥？

一般 join, filter 多的用 mysql. 整块数据读的用 mongo.

要是简单的话 plain text JSON 也不错。应该是 MongoDB 好一些，其设计之初就是为了应对大数据存储的。mongo

- 没有 schema 的严格定义，json 存取
- 爬虫的字段会经常变化，字段定义可能会变更，mongo 就对这方面很宽松
- mongo 是文档型的，天生为海量数据存储准备
- 可以很轻松的横向扩展，分片，复制集群分分钟

使用 mongo 也有坑，3.2 之后就换了新的 WiredTiger 引擎，占的内存略坑，对于没有太多 query 的存的数据库来看，内存还是会偶尔断片，没关系，在上面套一个 docker，还是一样很方便。

7、 后期数据处理（持久化、数据补全、归一化、格式化、转存、分类）。

爬虫数据持久化的几种常用方法:1、txt:用普通的磁盘 IO 操作即可,

2、csv

```
1 import csv
2 with open('xxx.csv','w') as f:
3     writer = csv.writer(f)
4     writer.writerow([])
5     writer.writerows([( ),( ),( )])
```

需注意单条数据写入的参数格式是列表，多条数据写入的参数格式是列表嵌套元组，推荐使用多条数据一次性写入，效率高。

3、json: 使用 json 模块中的 dump 函数

```
1 import json
2 data = {'xxx':'yyy'}
3 with open('zzz.json','w') as f:
4     json.dump(ob_data,f,ensure_ascii=False)
```

4、数据库：MySQL、MongoDB、Redis

存入 MySQL:

```
1 import pymysql
2 # __init__(self):
3 self.db = pymysql.connect('IP',... ...)
4 self.cursor = self.db.cursor()
5 # write_data(self):
6 self.cursor.execute('sql',[data1])
7 self.cursor.executemany('sql',[(data1),(data2),(data3)])
8 self.db.commit()
9 # main(self):
10 self.cursor.close()
11 self.db.close()
```

存入 MongoDB:

```

import pymongo
# __init__(self):
self.conn = pymongo.MongoClient('IP',27017)
self.db = self.conn['db_name']
self.myset = self.db['set_name']
# write_data(self):
self.myset.insert_one(dict)
# MongoDB - Command
>show dbs
>use db_name
>show collections
>db.collection_name.find().pretty()
>db.collection_name.count()
>db.collection_name.drop()
>db.dropDatabase()

```

存入 Redis:

使用 Redis 中的字符串、列表、哈希、集合、有序集合进行存储

处理丢失的数据:

处理丢失的数据

两种丢失的数据

数据的清洗

pandas 的拼接操作

pandas 的拼接分为两种:

级联: `pd.concat`, `pd.append`

合并: `pd.merge`, `pd.join`

使用 `pd.concat()` 级联

pandas 使用 `pd.concat` 函数, 与 `np.concatenate` 函数类似, 只是多了一些参数:

`objs`

`axis=0`

`keys`

`join='outer' / 'inner'`: 表示的是级联的方式, `outer` 会将所有的项进行级联 (忽略匹配和不匹配), 而 `inner` 只会将匹配的项级联到一起, 不匹配的不级联

`ignore_index=False`

不匹配级联

不匹配指的是级联的维度的索引不一致。例如纵向级联时列索引不一致, 横向级联时行索引不一致

有 2 种连接方式:

外连接: 补 `NaN` (默认模式)

内连接: 只连接匹配的项

使用 `pd.merge()` 合并

`merge` 与 `concat` 的区别在于, `merge` 需要依据某一共同的列来进行合并

使用 `pd.merge()` 合并时, 会自动根据两者相同 `column` 名称的那一列, 作为 `key` 来进行合并。

注意每一列元素的顺序不要求一致

参数:

`how`: `out` 取并集 `inner` 取交集

`on`: 当有多列相同的时候, 可以使用 `on` 来指定使用那一列进行合并, `on` 的值为一个列表

`key` 的规范化

当列冲突时, 即有多个列名称相同时, 需要使用 `on` 来指定哪一个列作为 `key`, 配合 `suffixes` 指定冲突列名

当两张表没有可进行连接的列时, 可使用 `left_on` 和 `right_on` 手动指定 `merge` 中左右两边的哪一列列作为连接的列

内合并与外合并: `out` 取并集 `inner` 取交集
内合并: 只保留两者都有的 `key` (默认模式)
外合并 `how='outer'`: 补 `NaN`

处理数据归一化数据规范化

- e) 为了消除指标之间的量纲和取值范围差异的影响, 需要进行标准化(归一化)处理, 将数据按照比例进行缩放, 使之落入一个特定的区域, 便于进行综合分析。
- f) 数据规范化方法主要有:
- g) - 最小-最大规范化
- h) - 零-均值规范化

归一化 (Normalization):

属性缩放到一个指定的最大和最小值(通常是 1-0)之间, 这可以通过 `preprocessing.MinMaxScaler` 类实现。

常用的最小最大规范化方法 $(x - \min(x)) / (\max(x) - \min(x))$

除了上述介绍的方法之外, 另一种常用的方法是将属性缩放到一个指定的最大和最小值(通常是 1-0)之间, 这可以通过 `preprocessing.MinMaxScaler` 类实现。

使用这种方法的目的包括:

- 1、对于方差非常小的属性可以增强其稳定性。
- 2、维持稀疏矩阵中为 0 的条目

标准化(Standardization):

将数据按比例缩放, 使之落入一个小的特定区间内, 标准化后的数据可正可负, 一般绝对值不会太大。

计算时对每个属性/每列分别进行

将数据按期属性(按列进行)减去其均值, 并处以其方差。得到的结果是, 对于每个属性/每列来说所有数据都聚集在 0 附近, 方差为 1。

使用 `z-score` 方法规范化 $(x - \text{mean}(x)) / \text{std}(x)$

这个在 `matlab` 中有特定的方程

使用 `sklearn.preprocessing.scale()` 函数, 可以直接将给定数据进行标准化:

使用 `sklearn.preprocessing.StandardScaler` 类, 使用该类的好处在于可以保存训练集中的参数(均值、方差)直接使用其对象转换测试集数据:

正则化:

正则化的过程是将每个样本缩放到单位范数(每个样本的范数为 1), 如果后面要使用如二次型(点积)或者其它核方法计算两个样本之间的相似性这个方法会很有用。

Normalization 主要思想是对每个样本计算其 p -范数, 然后对该样本中每个元素除以该范数, 这样处理的结果是使得每个处理后样本的 p -范数 ($l1\text{-norm}, l2\text{-norm}$) 等于 1。

p -范数的计算公式: $\|X\|_p = (|x_1|^p + |x_2|^p + \dots + |x_n|^p)^{1/p}$

该方法主要应用于文本分类和聚类中。例如, 对于两个 **TF-IDF** 向量的 $l2\text{-norm}$ 进行点积, 就可以得到这两个向量的余弦相似性。

- 1、可以使用 `preprocessing.normalize()` 函数对指定数据进行转换：
- 2、可以使用 `processing.Normalizer()` 类实现对训练集和测试集的拟合和转换：

python---爬虫[2]: 数据格式化

什么是 JSON

JSON 指的是 JavaScript 对象表示法 (JavaScript Object Notation)，是轻量级的文本数据交换格式，且具有自我描述性，更易理解。

JSON 看起来像 python 类型 (列表，字典) 的字符串。

数据处理

在之前的文章中，我们说到了怎么用 `response` 的方法，获取到网页正确解码后的字符串。如果还有不懂的，可以先阅读 [Python 爬虫 \(三\) Requests 库](#)。接下来以有道翻译为例子，说说怎么通过网页解码后的字符串，提取到翻译结果。

- `json.loads()`

首先我们先来看下，通过 `response.content.decode()` 解码之后得到的是什么类型的数据。通过打印出来 `type` 可以看到，我们解码之后得到的是 `str` 类型的数据。

若想进一步提取数据，则需要将 `str` 转换成 python 中的字典，python 的 `json` 库为我们提供了 `json.loads()` 用于将 `str` 类型的数据转成 `dict`。

再结合上述有道翻译的例子，得到字典类型的返回结果，并提取出来翻译结果。

`json.dumps()`

当我们将爬虫数据进行处理时，当然不希望它们仅仅是显示出来，还希望把提取到的数据保存起来。但是当我们在把字典类型的数据写入文本时，是写入不成功的。这个时候就需要将字典类型转换成字符串，再写入到文本之中，`json.dumps()` 的作用就是实现这一功能。

`json.loads()`、`json.load()`、`json.dump()` 以及 `json.dumps` 的区别

在输入 `json.loads()` 时，PyCharm 自动提示中，还有另外一个 `json.load()`。好奇之下，就整理了一下 `json.loads()`、`json.load()`、`json.dump()` 以及 `json.dumps` 的区别，感兴趣的可以通过 [Python 爬虫之 json.loads\(\)、json.load\(\)、json.dump\(\) 以及 json.dumps 的区别](#) 查看。

数据分类处理【重点】

数据聚合是数据处理的最后一步，通常是要使每一个数组生成一个单一的数值。

数据分类处理：

- 分组：先把数据分为几组
- 用函数处理：为不同组的数据应用不同的函数以转换数据
- 合并：把不同组得到的结果合并起来

数据分类处理的核心：

- `groupby()` 函数
- `groups` 属性查看分组情况
- eg: `df.groupby(by='item').groups`

高级数据聚合

使用 `groupby` 分组后，也可以使用 `transform` 和 `apply` 提供自定义函数实现更多的运算

- `df.groupby('item')['price'].sum() <==> df.groupby('item')['price'].apply(sum)`
- `transform` 和 `apply` 都会进行运算，在 `transform` 或者 `apply` 中传入函数即可
- `transform` 和 `apply` 也可以传入一个 `lambda` 表达式

例子：爬虫-美团各个分类的数据

最近兴趣使然，加上给朋友帮忙 研究了一下美团的爬虫，美团的数据都是做了反爬虫处理。

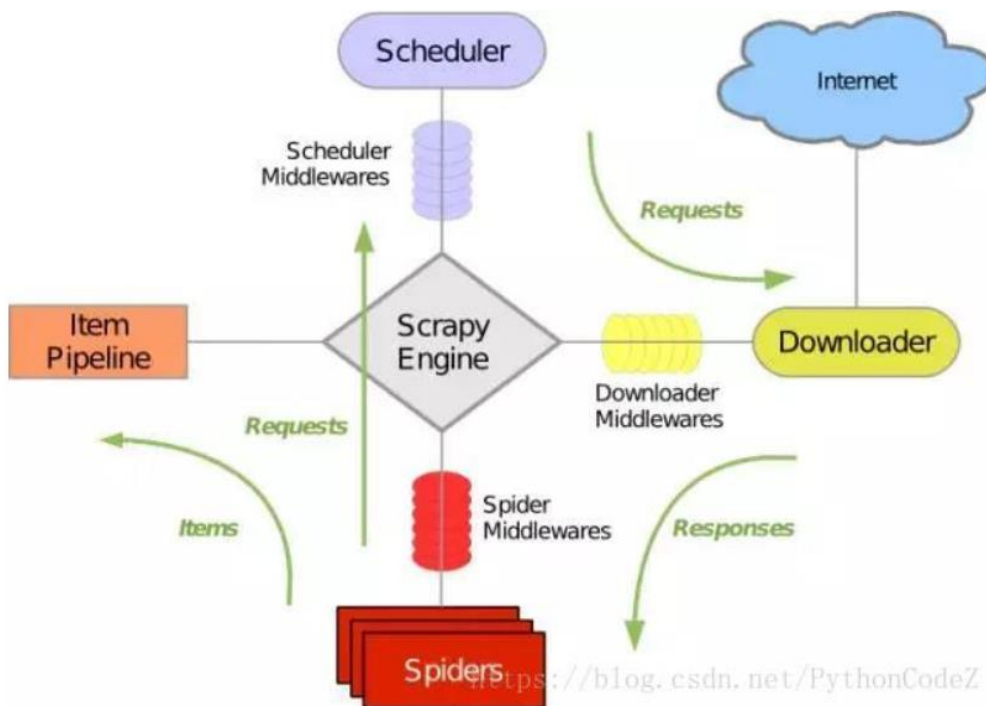
主要是通过验证码和访问 ip 限制 403 禁止访问的形式进行反爬虫处理。

但是通过处理，还是可以爬取一部分数据 目前有美团的爬取全国地区处理的爬虫程序，美团目前有很多分类，每个分类需要单独去处理，然后编写爬取规则，暂时只能爬取美团的美食，酒店，民宿，休闲娱乐。其他分类还在爬取当中。

爬虫 Scrapy 框架详解

i) 概述

下图显示了 Scrapy 的大体架构，其中包含了它的主要组件及系统的数据处理流程（绿色箭头所示）。下面就来一个个解释每个组件的作用及数据的过程。



j) 组件

2.1 Scrapy Engine（Scrapy 引擎）

Scrapy 引擎是用来控制整个系统的数据处理流程，并进行事务处理的触发。更多的详细内容可以看下面的数据处理流程。

2.2 Scheduler（调度）

调度程序从 Scrapy 引擎接受请求并排序列入队列，并在 Scrapy 引擎发出请求后返还给他们。

2.3 Downloader（下载器）

下载器的主要职责是抓取网页并将网页内容返还给蜘蛛 (Spiders)。

2.4 Spiders（蜘蛛）

蜘蛛是有 Scrapy 用户自己定义用来解析网页并抓取制定 URL 返回的内容的类，每个蜘蛛都能处理一个域名或一组域名。换句话说就是用来定义特定网站的抓取和解析规则。

蜘蛛的整个抓取流程（周期）是这样的：

首先获取第一个 URL 的初始请求，当请求返回后调取一个回调函数。第一个请求是通过调用 `startrequests()` 方法。该方法默认从 `starturls` 中的 `Url` 中生成请求，并执行解析来调用回调函数。在回调函数中，你可以解析网页响应并返回项目对象和请求对象或两者的迭代。这些请求也将包含一个回调，然后被 Scrapy 下载，然后有指定的回调处理。

在回调函数中，你解析网站的内容，同程使用的是 `Xpath` 选择器（但是你也可以使用 `BeautifulSoup`, `lxml` 或其他任何你喜欢的程序），并生成解析的数据项。

最后，从蜘蛛返回的项目通常会进驻到项目管道。

2.5 Item Pipeline（项目管道）

项目管道的主要责任是负责处理有蜘蛛从网页中抽取的项目，他的主要任务是清晰、验证和存储数据。当页面被蜘蛛解析后，将被发送到项目管道，并经过几个特定的次序处理数据。每个项目管道的组件都是有一个简单的方法组成的 `Python` 类。他们获取了项目并执行他们的方法，同时他们还需要确定的是是否需要在项目管道中继续执行下一步或是直接丢弃掉不处理。

项目管道通常执行的过程有：

清洗 HTML 数据

验证解析到的数据（检查项目是否包含必要的字段）

检查是否是重复数据（如果重复就删除）

将解析到的数据存储到数据库中

* 解释：引擎首先会将爬虫文件中的起始 `url` 获取，并且提交到调度器中。如果需要从 `url` 中下载数据，则调度器会将 `url` 通过引擎提交给下载器，下载器根据 `url` 去下载指定内容（响应体）。下载好的数据会通过引擎移交给爬虫文件，爬虫文件可以将下载的数据进行指定格式的解析。如果解析出的数据需要进行持久化存储，则爬虫文件会将解析好的数据通过引擎移交给管道进行持久化存储。

2.6 Downloader middlewares（下载器中间件）

下载中间件是位于 Scrapy 引擎和下载器之间的钩子框架，主要是处理 Scrapy 引擎与下载器之间的请求及响应。它提供了一个自定义的代码的方式来拓展 Scrapy 的功能。下载中间器是一个处理请求和响应的钩子框架。他是轻量级的，对 Scrapy 尽享全局控制的底层的系统。

2.7 Spider middlewares（蜘蛛中间件）

蜘蛛中间件是介于 Scrapy 引擎和蜘蛛之间的钩子框架，主要工作是处理蜘蛛的响应输入和请求输出。它提供一个自定义代码的方式来拓展 Scrapy 的功能。蜘蛛中间件是一个挂接到 Scrapy 的蜘蛛处理机制的框架，你可以插入自定义的代码来处理发送给蜘蛛的请求和返回蜘蛛获取的响应内容和项目。

2.8 Scheduler middlewares（调度中间件）

调度中间件是介于 Scrapy 引擎和调度之间的中间件，主要工作是处从 Scrapy 引擎发送到调度的请求和响应。他提供了一个自定义的代码来拓展 Scrapy 的功能。

k) 数据处理的流程

Scrapy 的整个数据处理流程有 Scrapy 引擎进行控制，其主要的运行方式为：

引擎打开一个域名，时蜘蛛处理这个域名，并让蜘蛛获取第一个爬取的 URL。

引擎从蜘蛛那获取第一个需要爬取的 URL，然后作为请求在调度中进行调度。

引擎从调度那获取接下来进行爬取的页面。

调度将下一个爬取的 URL 返回给引擎，引擎将他们通过下载中间件发送到下载器。

当网页被下载器下载完成以后，响应内容通过下载中间件被发送到引擎。

引擎收到下载器的响应并将它通过蜘蛛中间件发送到蜘蛛进行处理。

蜘蛛处理响应并返回爬取到的项目，然后给引擎发送新的请求。

引擎将抓取到的项目项目管道，并向调度发送请求。

系统重复第二部后面的操作，直到调度中没有请求，然后断开引擎与域之间的联系。

l) 驱动器

Scrapy 是由 Twisted 写的一个受欢迎的 Python 事件驱动网络框架，它使用的是非堵塞的异步处理。

面试题：如果最终需要将爬取到的数据值一份存储到磁盘文件，一份存储到数据库中，则应该如何操作 scrapy？

答：管道文件中的代码为：

#该类为管道类，该类中的 process_item 方法是用来实现持久化存储操作的。

```
class DoublekillPipeline(object):
```

```
    def process_item(self, item, spider):
        #持久化操作代码 （方式 1: 写入磁盘文件）
        return item
```

#如果想实现另一种形式的持久化操作，则可以再定制一个管道类：

```
class DoublekillPipeline_db(object):
```

```
    def process_item(self, item, spider):
        #持久化操作代码 （方式 1: 写入数据库）
        return item
```

在 settings.py 开启管道操作代码为：

#下列结构为字典，字典中的键值表示的是即将被启用执行的管道文件和其执行的优先级。

```
ITEM_PIPELINES = {
    'doublekill.pipelines.DoublekillPipeline': 300,
    'doublekill.pipelines.DoublekillPipeline_db': 200,
}
```

#上述代码中，字典中的两组键值分别表示会执行管道文件中对应的两个管道类中的 process_item 方法，实现两种不同形式的持久化操作。

五.Scrapy 发起 post 请求：

- 问题：在之前代码中，我们从来没有手动的对 start_urls 列表中存储的起始 url 进行过请求的发送，但是起始 url 的确是进行了请求的发送，那这是如何实现的呢？
- 解答：其实是因为爬虫文件中的爬虫类继承到了 Spider 父类中的 startrequests (self) 这个方法，该方法就可以对 starturls 列表中的 url 发起请求：

```
def start_requests(self):
    for u in self.start_urls:
        yield scrapy.Request(url=u, callback=self.parse)
```

【注意】该方法默认的实现，是对起始的 url 发起 get 请求，如果想发起 post 请求，则需要子类重写该方法。

- 重写 start_requests 方法，让其发起 post 请求：

```
def start_requests(self):
    #请求的 url
    post_url = 'http://fanyi.baidu.com/sug'
    # 表单数据
    formdata = {
        'kw': 'wolf',
    }
    # 发送 post 请求
    yield scrapy.FormRequest(url=post_url, formdata=formdata, callback=self.parse)
```

阅读目录

- [一.什么是 Scrapy?](#)
- [二.scrapy 安装](#)
- [三.基础使用](#)
- [四.scrapy 持久化操作：将爬取到糗百数据存储写入到文本文件中进行存储](#)
- [五.Scrapy 发起 post 请求：](#)

一.什么是 Scrapy?

Scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架，非常出名，非常强悍。所谓的框架就是一个已经被集成了各种功能（高性能异步下载，队列，分布式，解析，持久化等）的具有很强通用性的项目模板。对于框架的学习，重点是要学习其框架的特性、各个功能的用法即可。

二.scrapy 安装

Linux:

```
``pip3 install scrapy

**Windows: **
``a. pip3 install wheel
``b. 下载 twisted http:``/``/``www.lfd.uci.edu``/``~gohlke``/``pythonlibs``/``#twisted
``c. 进入下载目录，执行 pip3 install Twisted-``17.1``.``0``-cp35-cp35m-win_amd64.whl
``d. pip3 install pywin32
``e. ``pip3 install scrapy
```

强调：如果 windows10 安装 pip3 install Twisted-``17.1``.``0``-cp35-cp35m-win_amd64.whl 失败，请自行换成 32 位的即可解决，网上乱七八糟的答案请绕过，本人以亲测没什么卵用！！！！

提示：如果在 pycharm 中安装 scrapy 失败，两种解决办法：

- 1、把 pycharm 中的虚拟环境模式改成直接指向现在的 python 安装环境！

2、把 python 环境中的 scrapy, twisted 等直接复制到 pycharm 工程所在的虚拟环境中去！

三.基础使用

1.创建项目(cmd): scrapy startproject 项目名称

项目结构:

```
project_name/  
  scrapy.cfg:  
  project_name/  
    __init__.py  
    items.py  
    pipelines.py  
    settings.py  
    spiders/  
      __init__.py
```

scrapy.cfg 项目的主配置信息。（真正爬虫相关的配置信息在 settings.py 文件中）
items.py 设置数据存储模板，用于结构化数据，如：Django 的 Model
pipelines 数据持久化处理
settings.py 配置文件，如：递归的层数、并发数，延迟下载等
spiders 爬虫目录，如：创建文件，编写爬虫解析规则

2.创建爬虫应用程序(cmd):

cd project_name（进入项目目录）

scrapy genspider 应用名称 爬取网页的起始 url（例如：scrapy genspider qiubai www.qiushibaike.com）

3.编写爬虫文件:在步骤 2 执行完毕后，会在项目的 spiders 中生成一个应用名的 py 爬虫文件，文件源码如下：

应用名.py

```
# -*- coding: utf-8 -*-  
import scrapy
```

```
class QiubaiSpider(scrapy.Spider):  
    name = 'qiubai' #应用名称  
    #允许爬取的域名（如果遇到非该域名的 url 则爬取不到数据）  
    allowed_domains = ['https://www.qiushibaike.com/']  
    #起始爬取的 url  
    start_urls = ['https://www.qiushibaike.com/']
```

#访问起始 URL 并获取结果后的回调函数，该函数的 response 参数就是向起始的 url 发送请求后，获取的响应对象.该函数返回值必须为可迭代对象或者 NULL

```
def parse(self, response):  
    print(response.text) #获取字符串类型的响应内容  
    print(response.body)#获取字节类型的相应内容
```

4.设置修改 settings.py 配置文件相关配置:

#修改内容及其结果如下:

19 行: USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_0) AppleWebKit/537.36 (KHTML

ML, like Gecko) Chrome/68.0.3440.106 Safari/537.36' #伪装请求载体身份

22 行: ROBOTSTXT_OBEY = False #可以忽略或者不遵守 robots 协议

5.执行爬虫程序(cmd): scrapy crawl 应用名称

在 cmd 中执行爬虫程序时要先将目录切换到我们创建的 scrapy 项目目录下, 否则就会报: Unknown command: crawl 错误!!!!!!

小试牛刀: 将糗百首页中段子的内容和标题进行爬取

```
# -*- coding: utf-8 -*-
import scrapy
```

```
class QiubaiSpider(scrapy.Spider):
    name = 'qiubai' #应用名称
    allowed_domains = ['https://www.qiushibaike.com/']
    start_urls = ['https://www.qiushibaike.com/']

    def parse(self, response):
        #xpath 为 response 中的方法, 可以将 xpath 表达式直接作用于该函数中
        odiv = response.xpath('//div[@id="content-left"]/div')
        content_list = [] #用于存储解析到的数据
        for div in odiv:
            #xpath 函数返回的为列表, 列表中存放的数据为 Selector 类型的数据。我们解析到的内容被封装
            #在了 Selector 对象中, 需要调用 extract()函数将解析的内容从 Selector 中取出。
            author = div.xpath('.//div[@class="author clearfix"]/a/h2/text()')[0].extract()
            content=div.xpath('.//div[@class="content"]/span/text()')[0].extract()

            #将解析到的内容封装到字典中
            dic={
                '作者':author,
                '内容':content
            }
            #将数据存储在 content_list 这个列表中
            content_list.append(dic)

        return content_list
```

执行爬虫程序:

执行输出指定格式进行存储: 将爬取到的数据写入不同格式的文件中进行存储

```
scrapy crawl qiubai -o qiubai.json
scrapy crawl qiubai -o qiubai.xml
scrapy crawl qiubai -o qiubai.csv
```

四.scrapy 持久化操作: 将爬取到糗百数据存储写入到文本文件中进行存储

```
# -*- coding: utf-8 -*-
import scrapy
```

```
class QiubaiSpider(scrapy.Spider):
    name = 'qiubai'
    allowed_domains = ['https://www.qiushibaike.com/']
    start_urls = ['https://www.qiushibaike.com/']
```



```

def parse(self, response):
    #xpath 为 response 中的方法，可以将 xpath 表达式直接作用于该函数中
    odiv = response.xpath('//div[@id="content-left"]/div')
    with open('./data.txt', 'w') as fp:
        for div in odiv:
            #xpath 函数返回的为列表，列表中存放的数据为 Selector 类型的数据。我们解析到的内容
            #被封装在了 Selector 对象中，需要调用 extract() 函数将解析的内容从 Selector 中取出。
            author = div.xpath('..//div[@class="author clearfix"]/a/h2/text())[0].extract()

            content=div.xpath('..//div[@class="content"]/span/text())[0].extract()
            #持久化存储爬取到的内容
            fp.write(author + ':' + content + '\n')

```

注意：上述代码表示的持久化操作是我们自己通过 IO 操作将数据进行的文件存储。在 scrapy 框架中已经为我们专门集成好了高效、便捷的持久化操作功能，我们直接使用即可。要想使用 scrapy 的持久化操作功能，我们首先来认识如下两个文件：

items.py: 数据结构模板文件。定义数据属性。

pipelines.py: 管道文件。接收数据（items），进行持久化操作。

持久化流程：

- 1.爬虫文件爬取到数据后，需要将数据封装到 items 对象中。
- 2.使用 yield 关键字将 items 对象提交给 pipelines 管道进行持久化操作。
- 3.settings.py 配置文件中开启管道

小试牛刀：将糗事百科首页中的段子和作者数据爬取下来，然后进行持久化存储

爬虫文件：qiubaiDemo.py

```

# -*- coding: utf-8 -*-
import scrapy
from secondblood.items import SecondbloodItem

class QiubaideoSpider(scrapy.Spider):
    name = 'qiubaiDemo'
    allowed_domains = ['www.qiushibaike.com']
    start_urls = ['http://www.qiushibaike.com/']

    def parse(self, response):
        odiv = response.xpath('//div[@id="content-left"]/div')
        for div in odiv:
            # xpath 函数返回的为列表，列表中存放的数据为 Selector 类型的数据。我们解析到的内容被封
            #装在了 Selector 对象中，需要调用 extract() 函数将解析的内容从 Selector 中取出。
            author = div.xpath('..//div[@class="author clearfix"]//h2/text()).extract_first()

            author = author.strip('\n')#过滤空行
            content = div.xpath('..//div[@class="content"]/span/text()).extract_first()
            content = content.strip('\n')#过滤空行

            #将解析到的数据封装至 items 对象中
            item = SecondbloodItem()
            item['author'] = author
            item['content'] = content

```

```
yield item#提交 item 到管道文件 (pipelines.py)
```

items 文件: items.py

```
import scrapy
```

```
class SecondbloodItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    author = scrapy.Field() #存储作者
    content = scrapy.Field() #存储段子内容
```

管道文件: pipelines.py

```
# -*- coding: utf-8 -*-
```

```
# Define your item pipelines here
#
# Don't forget to add your pipeline to the ITEM_PIPELINES setting
# See: https://doc.scrapy.org/en/latest/topics/item-pipeline.html
```

```
class SecondbloodPipeline(object):
    #构造方法
    def __init__(self):
        self.fp = None #定义一个文件描述符属性
    #下列都是在重写父类的方法:
    #开始爬虫时, 执行一次
    def open_spider(self,spider):
        print('爬虫开始')
        self.fp = open('./data.txt', 'w')

    #因为该方法会被执行调用多次, 所以文件的开启和关闭操作写在了另外两个只会各自执行一次的方法中。
    def process_item(self, item, spider):
        #将爬虫程序提交的 item 进行持久化存储
        self.fp.write(item['author'] + ':' + item['content'] + '\n')
        return item

    #结束爬虫时, 执行一次
    def close_spider(self,spider):
        self.fp.close()
        print('爬虫结束')
```

配置文件: settings.py

```
#开启管道
```

```
ITEM_PIPELINES = {
    'secondblood.pipelines.SecondbloodPipeline': 300, #300 表示为优先级, 值越小优先级越高
}
```

Scrapy 递归爬取多页数据: 需求: 将糗事百科所有页码的作者和段子内容数据进行爬取并持久化存储

```
# -*- coding: utf-8 -*-
import scrapy
```

```

from qiushibaike.items import QiushibaikeItem
# scrapy.http import Request
class QiushiSpider(scrapy.Spider):
    name = 'qiushi'
    allowed_domains = ['www.qiushibaike.com']
    start_urls = ['https://www.qiushibaike.com/text/']

    #爬取多页
    pageNum = 1 #起始页码
    url = 'https://www.qiushibaike.com/text/page/%s/' #每页的 url

    def parse(self, response):
        div_list=response.xpath('//*[@id="content-left"]/div')
        for div in div_list:
            #//*[@id="qiushi_tag_120996995"]/div[1]/a[2]/h2
            author=div.xpath('.//div[@class="author clearfix"]//h2/text()').extract_first()
            author=author.strip('\n')
            content=div.xpath('.//div[@class="content"]/span/text()').extract_first()
            content=content.strip('\n')
            item=QiushibaikeItem()
            item['author']=author
            item['content']=content

            yield item #提交 item 到管道进行持久化

        #爬取所有页码数据
        if self.pageNum <= 13: #一共爬取 13 页（共 13 页）
            self.pageNum += 1
            url = format(self.url % self.pageNum)

            #递归爬取数据: callback 参数的值为回调函数（将 url 请求后，得到的相应数据继续进行 parse
            解析），递归调用 parse 函数
            yield scrapy.Request(url=url,callback=self.parse)

```

前言

学习 Scrapy 有一段时间了，当时想要获取一下百度汉字的解析，又不想一个个汉字去搜，复制粘贴太费劲，考虑到爬虫的便利性，这篇文章是介绍一个爬虫框架-Scrapy，非常主流的爬虫框架，写爬虫还不会 Scrapy，你就 out 啦?~

?爬虫的应用场景:

搜索多个汉字，存储下来汉字的解析

每隔一段时间获取一下最新天气，新闻等等

拿到豆瓣电影（豆瓣图书）的 top100 的电影名字、演员、上映时间以及各大网友的评论

需要下载网站的一系列图片，视频等，下载慕课网的课程视频

搜集安居客的所有房源，性价比分析

刷票、抢票

拿到微博当前的热门话题，自媒体需要即时写文章啦

安装

依赖环境:

Python 2.7 及以上

Python Package: pip and setuptools. 现在 pip 依赖 setuptools，如果未安装，则会自动安装 setuptools。

使用 pip 安装: `pip install Scrapy`

创建项目: `scrapy startproject [项目名]`

如创建 scrapy startproject qimairank, 会自动创建 Scrapy 的项目架构:

qimairank

```
--qimairank
  |--spiders
    |--__init__.py
  |--__init__.py
  |--items.py
  |--middlewares.py
  |--pipelines.py
  |--settings.py
--scrapy.cfg
```

scrapy.cfg: 项目的配置文件, 指定 settings 文件, 部署 deploy 的 project 名称等等。

qimairank: 项目的 python 模块。

spiders: 放置 spider 代码的目录。

items.py: 项目中的 item 文件。

pipelines.py: 项目中的 pipelines 文件。

middlewares.py: 项目的中间件。

settings.py: Scrapy 配置文件。更多配置信息查看: https://scrapy-chs.readthedocs.io/zh_CN/0.24/topics/settings.html

第一个爬虫: 爬取有道翻译

熟悉 Scrapy 框架后, 我们手写第一个爬虫, 爬取有道翻译的单词发音, 发音文件链接, 释义, 例句。

如单词 proportion: 有道翻译的详情连接为 <http://dict.youdao.com/w/eng/proportion>。本篇文章爬取的内容结果:

```
{"example": [{"en": "I seemed to have lost all sense of proportion.",
               "zh": "我好象已经丧失了有关比例的一切感觉。"},
              {"en": "The price of this article is out of(all) proportion to its value.",
               "zh": "这个商品的价格与它的价值完全不成比例。"},
              {"en": "But, the use of interception bases on the violation of the citizen rights, so it should be satisfactory of the principle of legal reservation and the principle of proportion.",
               "zh": "但是, 监听的适用是以侵害公民权利为前提的, 因此监听在刑事侦查中的运用必须满足法律保留原则和比例原则的要求。"}],
"explain": ["n. 比例, 占比; 部分; 面积; 均衡", "vt. 使成比例; 使均衡; 分摊"],
"pron": "[prə'pɔːʃ(ə)n]",
"pron_url": "http://dict.youdao.com/dictvoice?audio=proportion&type=1",
"word": "proportion"}
```

proportion

英 [prə'pɔːʃ(ə)n]  美 [prə'pɔːrʃən] 

n. 比例, 占比; 部分; 面积; 均衡

vt. 使成比例; 使均衡; 分摊

网络释义

专业释义

英英释义

— [数] 比例

托福词汇电子版-P ... proper a 适当的, 相当的; 正当的 **proportion** n
申请; 提议, 建议, 提出 ...

基于1219个网页-[相关网页](#)

+ 比率

+ 比

+ 比重

短语

Direct proportion 正比; [数] 正比例; 反比; 成正比

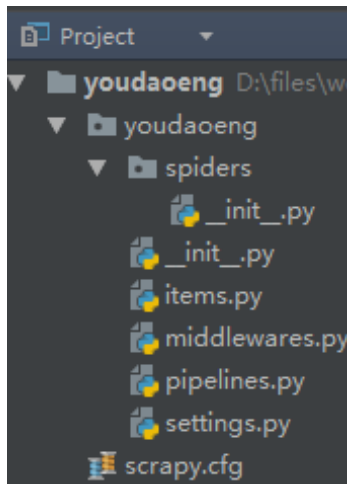
crown proportion 宽高比

continued proportion [数] 连比例; 连比

创建项目

在需要创建的目录下, scrapy startproject youdaoeng

回车即可创建默认的 Scrapy 项目架构。



创建 Item

创建 YoudaoengItem 继承 scrapy.Item，并定义需要存储的单词，发音，发音文件链接，释义，例句。

```
import scrapy

class YoudaoengItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    # 单词
    word = scrapy.Field()
    # 英式发音
    pron = scrapy.Field()
    # 发音 audio 文件链接
    pron_url = scrapy.Field()
    # 释义
    explain = scrapy.Field()
    # 例句
    example = scrapy.Field()
```

创建 Spider

在 spiders 目录下创建 EngSpider.py，并创建 class EngSpider，继承于 Spider。

```
from scrapy import Spider

class EngSpider(Spider):
    name = "EngSpider"
    # 允许访问的域
    allowed_domains = ["dict.youdao.com"]

    start_urls = [
        'http://dict.youdao.com/w/eng/agree', 'http://dict.youdao.com/w/eng/prophet',
        'http://dict.youdao.com/w/eng/proportion' ]

    def parse(self, response):
        pass
```

name: 用于区别 Spider, 该名字必须是唯一的。

starturls: Spider 在启动时进行爬取的 url 列表, 首先会爬取第一个。

def parse(self, response): 得到请求 url 后的 response 信息的解析方法。

有道翻译的网址为 <http://dict.youdao.com/>, 根据分析, 查询英文单词结果后链接更改, 如查询 agree, 跳转单词详情地址为 <http://dict.youdao.com/w/eng/agree>。所以几乎可以认为单词的详情页链接可以是 <http://dict.youdao.com/w/eng/> 拼接上单词本身, 所以配置 starturls 我们查询三个单词的释义详情。

解析

解析用的 Selectors 选择器有多种方法:

xpath(): 传入 xpath 表达式, 返回该表达式所对应的所有节点的 selector list 列表。

css(): 传入 CSS 表达式, 返回该表达式所对应的所有节点的 selector list 列表。

extract(): 序列化该节点为 unicode 字符串并返回 list。

re(): 根据传入的正则表达式对数据进行提取, 返回 unicode 字符串 list 列表。

下面我们用 xpath() 选择节点, xpath 的语法可参考 w3c 的

http://www.w3school.com.cn/xpath/xpath_nodes.asp 学习, 需要熟悉语法、运算符、函数等。

```
def parse(self, response):
    box = response.xpath('//*[ @id="results-contents"]')
    word = YoudaoengItem()
    # 简明释义
    box_simple = box.xpath('.//*[ @id="phrsListTab"]')
    # 判断查出来的字是否存在
    if box_simple:
        # 单词
        word['word'] = box_simple.xpath('.//h2[ @class="wordbook-js"]//span[ @class="keyword"]//text()').extract()[0]
        # 英式发音
        word['pron'] = box_simple.xpath(
            './/h2[ @class="wordbook-js"]//div[ @class="baav"]//*[ @class="phonetic"]//text()').extract()[0]
        # 发音链接
        word['pron_url'] = "http://dict.youdao.com/dictvoice?audio=" + word['word'] + "&type=1"
        # 释义
        word['explain'] = []
        temp = box_simple.xpath('.//div[ @class="trans-container"]//ul//li/text()').extract()
        for item in temp:
            if len(item) > 0 and not re.search(r'\n', item) and not re.match(r' ', item):
                print(item)
                word['explain'].append(item)
    # 例句
    time.sleep(3)
    word['example'] = []
    example_root = box.xpath('//*[ @id="bilingual"]//ul[ @class="ol"]/li')
    # 1. 双语例句是否存在
    if example_root:
        for li in example_root:
            en = ""
            for span in li.xpath('.//p[1]/span'):
                if span.xpath('.//text()').extract():
```

```

        en += span.xpath('./text()').extract()[0]
    elif span.xpath('./b/text()').extract():
        en += span.xpath('./b/text()').extract()[0]
    zh = str().join(li.xpath('./p[2]/span/text()').extract()).replace(' ',
''))
        word['example'].append(dict(en=en.replace('\n', '\\n'), zh=zh))
# 2.柯林斯英汉双解大辞典的例句是否存在
elif box.xpath('//*[id="collinsResult"]//ul[@class="ol"]//div[@class="examples
"]')):
    example_root = box.xpath('//*[id="collinsResult"]//ul[@class="ol"]//li')
    for i in example_root:
        if i.xpath('//*[class="exampleLists"]'):
            en = i.xpath(
                '//*[class="exampleLists"][1]//div[@class="examples"]/p[1]/te
xt()').extract()[0]
            zh = i.xpath(
                '//*[class="exampleLists"][1]//div[@class="examples"]/p[2]/te
xt()').extract()[0]
            word['example'].append(dict(en=en.replace('\n', '\\n'), zh=zh))
            if len(word['example']) >= 3:
                break
    yield word

```

最后 `yield word` 则是返回解析的 `word` 给 Item Pipeline，进行随后的数据过滤或者存储。

运行爬虫-爬取单词释义

运行爬虫，会爬取 `agree`、`prophet`、`proportion` 三个单词的详情，在项目目录下（`scrapy.cfg` 所在的目录）
`youdaoeng>scrapy crawl EngSpider -o data.json`

即可运行，窗口可以看见爬取的日志内容输出，运行结束后会在项目目录下生成一个 `data.json` 文件。


```

Example_spiders\youdaoeng scrapy crawl EngSpider -o data.json
2019-07-16 19:14:55 [scrapy.utils.log] INFO: Scrapy 1.5.1 started (bot: youdaoeng)
2019-07-16 19:14:55 [scrapy.utils.log] INFO: Versions: lxml 3.6.0.0, libxml2 2.9.5, cssselect 1.0.3, parsel 1.5.1, w3lib 1.19.0, Twisted 18.9.0, Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 10:07:46) [MSC v.1900 32 bit (Intel)]
2019-07-16 19:14:55 [scrapy.crawler] INFO: Overridden settings: {'BOT_NAME': 'youdaeng', 'FEED_URI': 'data.json', 'FEED_FORMAT': 'json', 'FEED_URI': 'data.json', 'NEWSPIDER_MODULE': 'youdaeng.spiders', 'ROBOTSTXT_OBEY': True, 'SPIDER_MODULES': ['eng.spiders']}
2019-07-16 19:14:55 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.corestats.CoreStats',
'scrapy.extensions.telnet.TelnetConsole',
'scrapy.extensions.feedexport.FeedExporter',
'scrapy.extensions.logstats.LogStats']
2019-07-16 19:14:56 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotStatsMiddleware',
'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
'scrapy.downloadermiddlewares.retry.RetryMiddleware',
'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
'scrapy.downloadermiddlewares.stats.DownloaderStats']
2019-07-16 19:14:56 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
'scrapy.spidermiddlewares.referrer.ReferrerMiddleware',
'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
'scrapy.spidermiddlewares.depth.DepthMiddleware']

```

```

2019-07-16 19:14:59 [scrapy.core.scraper] DEBUG: Scraped from <200 http://dict.youdao.com/w/eng/prophet>
{'example': [{ 'en': 'He is our prophet priest and king.',
                'zh': '他是我们的预言者牧师和国王。'},
              { 'en': 'And as its prophet, the Church celebrates you.',
                'zh': '并且作为它的先知，教会庆祝你。'},
              { 'en': 'There is no God but the one, true God and Muhammad is his Prophet.',
                'zh': '不存在上帝，但一个真正的上帝，穆罕默德是他的先知。'}],
'explain': ['n. 先知；预言者；提倡者'],
'pron': '["pr\u0252f\u026at"]',
'pron_url': 'http://dict.youdao.com/dictvoice?audio=prophet&type=1',
'word': 'prophet'}

```

生成的数据为所有 item 的 json 格式数组，中文字符都是 Unicode 编码，可通过一些在线的 json 解析网站如 <https://www.bejson.com/>，Unicode 转中文查看是我们想要的结果。

下载单词语音文件

单词读音的 mp3 链接为解析时候保存的 pronurl 字段，接下来我们下载单词 mp3 文件到本地。在 Item 下增加属性 pronsave_path，存储发音文件的本地地址：

```

# 发音 mp3 本地存放路径
pron_save_path = scrapy.Field()

```

并在 settings.py 文件中配置下载文件的目录，如在 D:\scrapy_files\目录下，则配置

```
FILES_STORE = "D:\\scrapy_files\\"
```

增加 ItemPipeline 重新发起文件下载请求：

```

class Mp3Pipeline(FilesPipeline):
    ...

    自定义文件下载管道
    ...

    def get_media_requests(self, item, info):
        ...

```

根据文件的 url 发送请求（url 跟进）

```
:param item:
:param info:
:return:
...
```

meta 携带的数据可以在 response 获取到

```
yield scrapy.Request(url=item['pron_url'], meta={'item': item})
```

```
def item_completed(self, results, item, info):
    ...
```

处理请求结果

```
:param results:
:param item:
:param info:
:return:
...
```

```
file_paths = [x['path'] for ok, x in results if ok]
if not file_paths:
    raise DropItem("Item contains no files")
```

```
# old_name = FILES_STORE + file_paths[0]
# new_name = FILES_STORE + item['word'] + '.mp3'
```

文件重命名（相当于剪切）

```
# os.rename(old_name, new_name)
```

```
# item['pron_save_path'] = new_name
```

返回的 result 是除去 FILES_STORE 的目录

```
item['pron_save_path'] = FILES_STORE + file_paths[0]
return item
```

```
def file_path(self, request, response=None, info=None):
    ...
```

自定义文件保存路径

默认的保存路径是在 FILES_STORE 下创建的一个 full 来存放，如果我们想要直接在 FILES_STORE 下存放，则需要自定义存放路径。

默认下载的是无后缀的文件，需要增加.mp3 后缀

```
:param request:
:param response:
:param info:
:return:
...
```

```
file_name = request.meta['item']['word'] + ".mp3"
return file_name
```

需要更改 settings.py 文件，配置 Mp3Pipeline，后面的 300 为优先级，数字越大，优先级越低。

```
# Configure item pipelines
```

```
# See https://doc.scrapy.org/en/latest/topics/item-pipeline.html
```

```
ITEM_PIPELINES = {
    # 'youdaoeng.pipelines.YoudaoengPipeline': 300,
    'youdaoeng.pipelines.Mp3Pipeline': 300,
}
```

运行

```
youdaoeng>scrapy crawl EngSpider -o data1.json
```

等待运行完成，则在项目目录下生成了 data1.json，并在 D:\scrapy_files\目录下生成了我们爬取的三个单词的释义。

项目源码

数据分析：

1. 科学运算函数库（SciPy 和 NumPy 常用运算）。
2. 数据分析库（Pandas 中封装的常用算法）。
3. 常用的模型及对应的场景（分类、回归、聚类）。
4. 提取了哪些具体的指标。
5. 如何评价模型的优劣。
6. 每种模型实际操作步骤，对结果如何评价。

项目相关：

1. 项目团队构成以及自己在团队中扮演的角色（在项目中的职责）。
2. 项目的业务架构（哪些模块及子模块）和技术架构（移动端、PC 端、后端技术栈）。
3. 软件控制管理相关工具（版本控制、问题管理、持续集成）。
4. 核心业务实体及其属性，实体与实体之间的关系。
5. 用到哪些依赖库，依赖库主要解决哪方面的问题。
6. 项目如何部署上线以及项目的物理架构（Nginx、Gunicorn/uWSGI、Redis、MongoDB、MySQL、Supervisor 等）。
7. 如何对项目进行测试，有没有做过性能调优。
8. 项目中遇到的困难有哪些，如何解决的。

项目相关：

- 1、项目团队构成以及自己在团队中扮演的角色（在项目中的职责）。

你公司 IT 部门的人在干什么（信息部岗位职责）

介绍信息化那些事儿，欢迎关注，携手致远。微信公众号同名。

信息化时代，企业的 IT 岗位必不可少，成熟的企业都会有自己的 IT 部（信息管理部），该部门岗位通常分为 CIO、IT 总监、IT 经理、专业岗位人员这四类，下面介绍下各岗位的工作情况。

一、CIO（首席信息官）

1、理解公司战略

CIO 是公司战略信息化支撑的第一责任人，只有充分理解公司战略，才能统筹公司内外资源去落地支撑战略的信息化需求。

理解公司战略，是信息化的“梯子”搭在什么墙上的问题，要是梯子搭错墙，下面兄弟在怎么拼命，都白费功夫。

2、IT 规划

适合自己的才是最好的，在理解公司战略的基础上通过 IT 规划，建设与公司当前业务规模、未来一段时间业务发展规划相匹配的信息化系统。这里面涉及到建设什么样的系统，用什么样的技术架构，自建还是找外部供应商，如何与现有信息化资源整合等系列问题，这些问题都需要 CIO 综合考虑。

3、横向部门 leader 对接

信息部门归根结底是为管理、为业务服务的，系统是否有价值，很大程度上是业务部门说了算。花了大价钱把系统建好，最后发现业务部门不认可，那就是人为系统打工，而不是系统为人服务。这里面业务部门的认可，两个部门负责人建立良好的互动关系是桥梁。

4、供应商资源整合

信息化系统建设通常是建的时候最耗费资源，建成之后就耗费较少了。公司为了控制信息化成本，信息化资源肯定不会高配，因为成本等在内的众多因素，公司肯定需要请外部供应商资源进来建设信息化系统，IT 部门假如有几个靠谱、给力的供应商，系统建设速度与质量肯定上去。话说钱虽然重要，但 CIO 对各供应商能力方向与水平把握程度、与供应商的连接关系也同样至关重要。

5、部门员工管理

正常的部门管理工作，在此不过多阐述。想说的一点是，信息部内部管理风格与水平除了依赖于部门负责人外，其实受公司整体管理机制、企业文化影响更大。

二、IT 总监

公司内部的 IT 总监，是某项垂直领域解决方案的专家，可用说是“IT 里面最懂业务的，业务里面最懂 IT 的人”，这里说的业务是某一项或几项业务。若是开发人员的话，那就是开发里面的架构师，能够独自带领一帮开发兄弟从 0 到 1 搞定一个大型信息化系统。

IT 总监是 CIO 的信息化规划信息支撑、落地的最好帮手，是各类信息化项目的“项目集经理”，合格的 IT 总监能够上下管理得当，把自己负责各领域的信息化项目稳步推进，并对系统价值实现逻辑了如指掌。

IT 总监通常不直接处理事务性工作，但需要带领团队完成部门分派的任务，对各项工作交付物的关键节点进行质量把控。

PS：规模不大的公司，IT 总监的工作通常就是部门负责人兼顾了。

三、IT 经理

1、IT 项目管理

IT 经理通常是信息化项目经理，对于有供应商参与进来的项目，那就是“甲方项目经理”，项目成败的负责人、项目整体成败的考核对象。需要其具备一定的项目管理知识，项目前期需要评估项目成本、价值点、采购选型等。过程中需要协调业务部门、部门内部同事、供应商等各方面资源，把项目按合理的节奏推进，并把控项目交付质量。

2、解决方案构建/审核

负责项目过程中的解决方案、产品设计方案输出，有些需要 IT 经理自己干，有些是供应商资源来做但也需要 IT 经理把关并确认，假如项目经理解决方案、产品设计方面能力不足，项目很容易出现返工或进度拖延的情况。好的 IT 经理，能最终确保按时交付的系统满足业务各层级的需求。

3、公司信息化系统运营

系统建设完成后，如何最大化系统价值，是 IT 经理必须考虑的事情。首先，需要 IT 经理联合业务骨干、系统关键用户，把系统推广出去，让系统能够帮助到更多用户。其次，根据业务应用系统的反馈，持续组织资源对系统功能进行优化完善。最后，最重要的是，分析所沉淀的业务数据，为业务与管理提供数据服务。

四、专业岗位人员

在 IT 部门，从工作分工的专业类别来说，通常会分为以下几类专业：

- **开发人员：**系统开发、系统优化、数据分析等工作；这部分工作可以交给供应商来做，不过稍微大点的公司，为确保需求响应速度，都会有自己的开发人员，只是多少的问题。
- **系统管理人员：**负责权限配置、流程配置、系统异常登记与处理跟进、常见问题咨询等工作；通常会分为办公系统类（OA、邮箱、IM 等通用办公系统）、业务应用类（ERP、财务、BI 等专业类）。
- **基础架构管理人员：**负责服务器、网络、存储资源、网络安全管理，包括相关资源采购需求评估、资源分配、数据备份等工作。
- **桌面运维人员：**负责公司员工日常办公软件安装、电脑初始化、办公电脑问题处理等各类工作。

以上各类工作，会从专业程度、内部需求工作饱和度、成本、信息安全等各方面因素，综合考虑是公司内部招聘员工，还是寻找相关外部供应商提供服务。

虽然是说普通/基层专业人员，但考虑到各公司规模、人员编制数量、成本等因素，部分公司是和 IT 经理岗位职责一并考虑的，会把不同类型的工作根据员工的能力进行综合分配。

发布于 2020-04-01

- 2、 项目的业务架构（哪些模块及子模块）和技术架构（移动端、PC 端、后端技术栈）。
- 3、 软件控制管理相关工具（版本控制、问题管理、持续集成）。
- 4、 核心业务实体及其属性，实体与实体之间的关系。
- 5、 用到哪些依赖库，依赖库主要解决哪方面的问题。
- 6、 项目如何部署上线以及项目的物理架构（Nginx、Gunicorn/uWSGI、Redis、MongoDB、MySQL、Supervisor 等）。

[线上环境部署 Django, nginx+uwsgi 和 nginx+gunicorn, 这两种方案, 应该如何选择?]

大家是采用的何种部署方式?

第一种,高并发稳定一点

我们公司使用的是 nginx+gunicorn,主要是方便。性能可以从其他方面优化。

随便吧

我们用的是 `nginx supervisor gunicorn`

Instagram 由 `uwsgi` 转到 `gunicorn`，建议用 `gunicorn`，配置简单方便。

推荐 `nginx supervisor gunicorn`

配置简单，运维方便。

用的 `nginx+gunicorn` 方式，`uwsgi` 没用过所以没法对比，就 `gunicorn` 的感受也来讲已经很快了，`nginx` 处理掉了几乎全部的静态文件请求，实际上需要 `gunicorn` 再来处理的请求已经很少了。

`gunicorn` 可以用 `Python` 文件直接配置，试用起来比较舒服。

我觉得这两种相差不多，根据你们相关的运维人员和开发人员的熟悉程度来决定。

现在的网站其实大部分处理的都是静态文件请求，除了诸如秒杀活动等等特定的业务，一般业务的请求量并不是很大。

所以你可以根据：1、你们当前的业务与以后可能会增加的业务；2、你们的运维人员的技能来决定，哪个更熟悉就选哪个

[如何使用 `Nginx` 和 `uWSGI` 或 `Gunicorn` 在 `Ubuntu` 上部署 `Flask Web` 应用]

我在很多的博客中都看过有关 `Flask` 应用的部署，也有很多博主在开博后都记录了部署的教程，因为其中的坑可以说不少。一开始我在网上看到相比较与 `Ubuntu`，`CentOS` 因为更新少作为服务器的操作系统会更加稳定。所以在第一次购买云服务器时，我选择了 `CentOS`，后来由于 `CentOS` 不同发行版的 `Nginx` 缘故，我又换成了 `Ubuntu` 的镜像

首先呢，我们先来了解下关于 `Web` 服务器与 `Web` 应用还有 `WSGI` 之间的联系

一、介绍

`WSGI`（`Web Server Gateway Interface`），翻译为 `Python web` 服务器网关接口，即 `Python` 的 `Web` 应用程序（如 `Flask`）和 `Web` 服务器（如 `Nginx`）之间的一种通信协议。也就是说，如果让你的 `Web` 应用在任何服务器上运行，就必须遵循这个协议。

那么实现 `WSGI` 协议的 `web` 服务器有哪些呢？就比如 `uWSGI` 与 `gunicorn`。两者都可以作为 `Web` 服务器。可能你在许多地方看到的都是采用 `Nginx + uWSGI`（或 `gunicorn`）的部署方式。实际上，直接通过 `uWSGI` 或 `gunicorn` 直接部署也是可以让外网访问的，那你可能会说，那要 `Nginx` 何用？别急，那么接下来介绍另一个 `Web` 服务器——`Nginx`

`Nginx` 作为一个高性能 `Web` 服务器，具有负载均衡、拦截静态请求、高并发...等等许多功能，你可能要问了，这些功能和使用 `Nginx + WSGI` 容器的部署方式有什么关系？

首先是负载均衡，如果你了解过 [OSI 模型](#) 的话，其实负载均衡器就是该模型中 4~7 层交换机中的一种，它的作用是能够仅通过一个前端唯一的 `URL` 访问分发到后台的多个服务器，这对于并发量非常大的企业级 `Web` 站点非常有效。在实际应用中我们通常会让 `Nginx` 监听（绑定）80 端口，通过多域名或者多个 `location` 分发到不同的后端应用。

其次是拦截静态请求，简单来说，`Nginx` 会拦截到静态请求（静态文件，如图片），并交给自己处理。而动态请求内容将会通过 `WSGI` 容器交给 `Web` 应用处理；

`Nginx` 还有其他很多的功能，这里便不一一介绍。那么前面说了，直接通过 `uWSGI` 或 `gunicorn` 也可以让外网访问到的，但是鉴于 `Nginx` 具有高性能、高并发、静态文件缓存、及以上两点、甚至还可以做到限流与访问控制，所以选择 `Nginx` 是很有必要的；

这里可以说明，如果你选择的架构是：`Nginx + WSGI 容器 + web 应用`，`WSGI` 容器相当于一个中间件；如果选择的架构是 `uWSGI + web 应用`，`WSGI` 容器则为一个 `web` 服务器

二、实际部署：

这篇部署的教程是在你已经购买好虚拟主机，并且已经搭建好开发环境的前提下进行的，如果你还没有搭建好开发环境，可以参考我写的文档：

[阿里云 Ubuntu 云服务器上搭建 Python 和 Flask 的开发环境](#)

普遍的部署方式都是通过让 Nginx 绑定 80 端口，并接受客户端的请求将动态内容的请求反向代理给运行在本地端口的 uWSGI 或者 Gunicorn，所以既可以通过 Nginx + uWSGI 也可以通过 Nginx + Gunicorn 来部署 Flask 应用，这篇教程中都将一一介绍这两种方法

当然采用不同的 WSGI 容器，Nginx 中的配置也会有所不同

1. Nginx + uWSGI:

1.1 配置 uWSGI:

我们现在虚拟环境下安装好 uWSGI:

```
(venv) $ pip install uwsgi
```

安装完成之后我们在项目的目录下（即你实际创建的 Flask 项目目录，在本文所指的项目目录都假设为 /www/demo）创建以 .ini 为扩展名的配置文件。在设置与 Nginx 交互的时候有两种方式：

第一种是通过配置网络地址，第二种是通过本地的 .socket 文件进行通信。需要注意的是，不同的交互方式下，Nginx 中的配置也会有所不同

如果采用的是第一种网络地址的方式，则将之前创建 uwsgi.ini 配置文件添加如下的配置内容：

```
[uwsgi]
socket = 127.0.0.1:8001 //与nginx通信的端口
chdir = /www/demo/ //你的Flask项目目录
wsgi-file = run.py
callable = app //run.py文件中flask实例化的对象名
processes = 4 //处理器个数
threads = 2 //线程个数
stats = 127.0.0.1:9191 //获取uwsgi统计信息的服务地址
```

这里的 wsgi-file 参数所指的 run.py 其实是启动文件，你也可以使用 manage.py。不过我通常习惯创建一个这样的文件，可以直接运行该文件来启动项目：

```
from app import app
if __name__ == '__main__':
    app.run()
```

保存好配置文件后，就可以通过如下的命令来启动应用了：

```
(venv) $ uwsgi uwsgi.ini
```

如果你采用的是第二种本地 socket 文件的方式，则添加如下的配置内容：

```
[uwsgi]
socket = /www/demo/socket/nginx_uwsgi.socket //与nginx通信的socket文件
chdir = /www/demo/
wsgi-file = run.py
callable = app
processes = 4
threads = 2
stats = 127.0.0.1:9191
```

可以看到，其实与网络地址的配置方式只有 `socket` 参数的配置不同，在这里填写好路径名和文件名并启动 `uWSGI` 后，将会自动在改目录下生成 `nginx_uwsgi.socket` 文件，这个文件就是用来与 `Nginx` 交互的。

1.2 配置 Nginx

首先我们来通过 `apt` 安装 `Nginx`:

```
$ sudo apt-get install nginx
```

安装完成之后，我们 `cd` 到 `/etc/nginx/` 的目录下（可能由于不同系统导致不同的 `Nginx` 发行版缘故，目录有所差别，在此只针对 `Ubuntu` 中的发行版的 `Nginx`），可以看到 `Nginx` 的所有配置文件。

其中 `nginx.conf` 文件为主配置文件，可以用来修改其全局配置；`sites-available` 存放你的配置文件，但是在这里添加配置是不会应用到 `Nginx` 的配置当中，需要软连接到同目录下的 `sites-enabled` 当中。但是在我实际操作的过程中中，当我在 `sites-available` 修改好配置文件后，会自动更新到 `sites-enabled`。如果没有的话，则需要像上述的操作那样，将修改好的配置文件软链接到 `sites-enabled` 当中

在上边说到，配置 `uWSGI` 有两种与 `Nginx` 交互的方式，那么选择不同的方式的话在 `Nginx` 的配置也会有所不同：

第一种：网络配置方式：

这里的 `proxy_set_header` 设置的三个参数的作用都是能够直接获得到客户端的 `IP`，如果你感兴趣可以参考：[Nginx 中 proxysetheader 理解](#)

用 `include uwsgi_params` 导入 `uWSGI` 所引用的参数，通过 `uwsgi_pass` 反向代理给在 `localhost:8001` 运行的 `uWSGI`:

```
server {
    listen 80; # 监听的端口号
    root /www/demo; #Flask 的项目目录
    server_name xxx.xx.xxx.xxx; # 你的公网ip 或者域名
    location / {
        proxy_set_header x-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        include uwsgi_params;
        uwsgi_pass localhost:8001;
    }
    # 配置static 的静态文件:
    location ~ ^\/static\/.*$ {
        root /www/demo; # 注意! 这里不需要再加/static 了
    }
}
```

在每次完 `Nginx` 配置文件内容后，需要通过如下的命令来重启 `Nginx`:

```
$ nginx -s reload
```

第二种：socket 文件方式：

与上边的配置内容大体相同，只是在配置 `uwsgi_pass` 不是反向代理给网络地址，而是通过 `socket` 文件进行交互，我们只需要指定之前设置的路径和文件名即可：


```
server {
    listen 80;
    root /www/demo;
    server_name xxx.xx.xxx.xxx;
    location / {
        proxy_set_header x-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        include uwsgi_params;
        uwsgi_pass unix:/www/demo/socket/nginx_uwsgi.socket;
    }
    location ~ ^/\bstatic\b/.*$ {
        root /www/demo;
    }
}
```

2. Nginx + Gunicorn

2.1 配置 Gunicorn:

首先在虚拟环境下安装 Gunicorn:

```
(venv) $ pip install gunicorn
```

安装完成后, 我们来创建以 .py 结尾的配置文件, 这里我参考了 Jiyuankai 的 [GitHub](#) 关于 Gunicorn 的配置文件内容:

```
from gevent import monkey
monkey.patch_all()
import multiprocessing
debug = True
loglevel = 'debug'
bind = '127.0.0.1:5000' //绑定与 Nginx 通信的端口
pidfile = 'log/gunicorn.pid'
logfile = 'log/debug.log'
workers = multiprocessing.cpu_count() * 2 + 1
worker_class = 'gevent' //默认为阻塞模式, 最好选择 gevent 模式
```

需要注意的是要在配置文件的同层目录下创建 log 文件, 否则运行 gunicorn 将报错。添加完配置内容并保存为 gconfig.py 文件后, 我们就也可以通过 gunicorn 来运行 Flask 应用了:

```
(venv)$ gunicorn -c /www/demo/gconfig.py run:app
```

2.2 配置 Nginx:

和 uWSGI 的任意一种配置方法类似, 只是在 location 中的配置有所不同:

```
server {
    listen 80;
    root /www/demo;
    server_name xxx.xx.xxx.xxx;
    location / {
        proxy_set_header x-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_pass http://localhost:5000/; # gunicorn 绑定的端口号
    }
    # 配置 static 的静态文件:
```

```

        location ~ ^\/static\/.*$ {
            root /www/demo;
        }
    }
}

```

通过 Gunicorn 的 Nginx 配置中，我们只需要通过 `proxy_pass` 参数反向代理给运行在 `http://localhost:5000/` 上的 Gunicorn

三、守护进程

如果你采取如上的任意一种部署方式，在 Nginx 与 uWSGI 或 Gunicorn 同时运行，并且配置无误的状态下，那么你现在应该是可以通过你的公网 ip 或者域名访问到你的网站了。

但是还有一个问题，到目前为止，uWSGI 和 gunicorn 都是直接通过命令行运行，并不能够在后台运行，也是当我们关闭了 xShell（或者你使用的是 Putty 及其他 SSH 连接的软件），将无法再访问到你的应用。所以我们需要让 uWSGI 或 gunicorn 在后台运行，也就是所谓的 **daemon**（守护进程）。

1. nohup:

如果你熟悉 Linux 命令，你应该知道在 Linux 中后台运行可以通过 `nohup` 命令，例如我们要让 gunicorn 在后台运行，我们只需要运行 `nohup` 命令：

```
(venv) $ nohup gunicorn -c gconfig.py run:app &
```

运行后你可以通过 `ps -e | grep gunicorn` 指令来查看到当前 gunicorn 的运行状态：

image

如果你选择的是 uWSGI，同样也可以通过 `nohup` 命令来实现守护进程：

```
(venv) $ nohup uwsgi uwsgi.ini &
```

这样你就可以关闭连接服务器的终端，还能通过你的浏览器访问到你的 Flask 应用了！

2. supervisor

但是 `nohup` 运行的后台程序并不能够可靠的在后台运行，我们最好让我们的后台程序能够监控进程状态，还能在意外结束时自动重启，这就可以使用一个使用 Python 开发的进程管理程序 **supervisor**。

参考：<https://www.cnblogs.com/Dicky-Zhang/p/6171954.html>

首先我们通过 `apt` 来安装 supervisor：

```
$ sudo apt-get install supervisor
```

安装完成后，我们在 `/etc/supervisor/conf.d/` 目录下创建我们控制进程的配置文件，并以 `.conf` 结尾，这样将会自动应用到主配置文件当中，创建后添加如下的配置内容：

```

[program:demo]
command=/www/demo/venv/bin/gunicorn -c /pushy/blog/gconfig.py run:app
directory=/www/demo //项目目录
user=root
autorestart=true //设置自动重启
startretires=3 //重启失败 3 次

```

在上面的配置文件中，[program:demo]设置了进程名，这与之后操作进程的状态名称有关，为demo;command 为进程运行的命令，必须使用绝对路径，并且使用虚拟环境下的 gunicorn 命令；user 指定了运行进程的用户，这里设置为 root

保存配置文件之后，我们需要通过命令来更新配置文件：

```
$ supervisorctl update
```

命令行将显示：demo: added process group，然后我们来启动这个 demo 进程：

```
$ supervisorctl start demo
```

当然你也直接在命令行输入 supervisorctl 进入 supevisor 的客户端，查看到当前的进程状态：

```
demo          RUNNING    pid 17278, uptime 0:08:51
```

通过 stop 命令便可以方便的停止该进程：

```
supervisor> stop demo
```

- 7、 如何对项目进行测试，有没有做过性能调优。
- 8、 项目中遇到的困难有哪些，如何解决的。