PTSD-追光者

昵称： PTSD-追光者
园龄： 2年9个月
粉丝： 2
关注： 27
+加关注

< 2021年4月 >

日	一	二	三	四	五	六
28	29	30	31	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	1
2	3	4	5	6	7	8

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

随笔档案
2020年4月(1)
2019年8月(1)
2019年6月(3)
2019年5月(1)
2019年4月(4)
2018年7月(1)

阅读排行榜
1. 浅谈工作中celery与Redis遇到的一些问题(3532)
2. 浅谈如何提高网站的并发量（性能）(1026)

浅谈工作中celery与Redis遇到的一些问题

celery的内存泄漏?

总结： celery执行完任务不释放内存与原worker一直没有被销毁有关，因此CELERYD_MAX_TASKS_PER_CHILD可以适当配置小点，而任务并发数与CELERYD_CONCURRENCY配置项有关，

每增加一个worker必然增加内存消耗，同时也影响到一个worker何时被销毁，因为celery是均匀调度任务至每个worker，因此也不宜配置过大，适当配置。

CELERYD_MAX_TASKS_PER_CHILD

CELERYD_CONCURRENCY = 20 # 并发worker数

CELERYD_FORCE_EXECV = True # 非常重要,有些情况下可以防止死锁

CELERYD_MAX_TASKS_PER_CHILD = 100 # 每个worker最多执行万100个任务就会被销毁，可防止内存泄露

CELERYD_TASK_TIME_LIMIT = 60 # 单个任务的运行时间不超过此值，否则会被SIGKILL 信号杀死

任务发出后，经过一段时间还未收到acknowledge , 就将任务重新交给其他worker执行

CELERY_DISABLE_RATE_LIMITS = True

在Django中使用celery内存泄漏问题?

在django下使用celery作为异步任务系统，十分方便。

同时celery也提供定时任务机制，celery beat。使用celery beat 可以为我们提供 cron，schedule 形式的定时任务。

在django下使用celery beat的过程中，发现了 celery beat进程 占用内存非常大，而且一直不释放。

怀疑其有内存占用不释放的可能。

因为之前使用django的时候，就知道在django中开启DEBUG模式，会为每次的SQL查询 缓存结果。 celery beat 作为 定时任务的timer和heartbeat程序，是长期运行的，而我使用了MYSQL作为存储定时任务的backend。

因为每次heartbeat和timer产生的sql查询在开启了DEBUG模式下的django环境中，都会缓存查询结果集。因此 celery beat占用的 内存会一直不释放。在我的线上环境中 达到10G内存占用!

解决： 关掉django的DEBUG模式，在setting中，设置DEBUG=False 即可。 关闭DEBUG模式后的celery beat程序 的内存占用大概 一直维持在150M左右。

数据库连接是单利吗? 有必要实现多例吗?

单例数据库连接在连接池中只有一个实例，对系统资源的开销比较少，甚至可以长时间保持连接不回收，以节省创建连接和回收连接的时间。

但这样的连接在多用户并发时不能提供足够的效率，形象的来讲就是大家要排队。 初级程序员的作法是每个用户需求过来都打开一次连接，用完回收掉。

进阶的做法是建立一个连接池，连接池里面给定一些已打开的连接，用程序控制这些连接的分配与调度

数据库链接用单例模式的原因：

单例只保留一个对象，可以减少系统资源开销。

3. python(311)

4. 网络编程,并发编程(264)

5. MySQL(261)

提高创建速度，每次都获取已经存在的对象因此提高创建速度全局共享对象。

单例在系统中只存在一个对象实例，因此任何地方使用此对象都是一个对象避免多实例创建使用时产生的逻辑错误。

例模式是一种常用的软件设计模式，它的核心结构只包含一个被称为单例的特殊类。它的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

单例模式有3种实现方式：懒汉式、饿汉式和双重锁的形式。

单例模式优点 只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等 因为类控制了实例化过程，所以类可以灵活更改实例化过程

单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。 单例的缺点 就是不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，

单例就会引起数据的错误，不能保存彼此的状态。 用单例模式，就是在适用其优点的状态下使用。

比如： 要进一个房间（数据库），就为这个房间开了一扇门（数据库类），一般情况下是一个人开一扇门， 不管你进出（数据库操作）这个房间多少次，门就这一扇（单例），

当然一个人也可以开很多扇门（非单例）， 但你知道一个房间能开的门的数量是有限的，因此你不使用单例的话，一是性能慢一些，二是走别人的门，让别人无门可进。。。

数据类型的堆栈存储?

堆栈是一个后进先出的数据结构，其工作方式就像一堆汽车排队进去一个死胡同里面，最先进去的一定是最后出来。

队列是一种先进先出的数据类型，它的跟踪原理类似于在超市收银处排队，队列里的的第一个人首先接受服务，新的元素通过入队的方式添加到队列的末尾，而出队就是将队列的头元素删除。

栈：是一种容器，可存入数据元素、访问元素、删除元素

特点：只能从顶部插入（入栈）数据和删除（出栈）数据

原理：LIFO(Last In First Out)后进先出 栈可以使用顺序表实现也可使用链表实现 使用python列表实现代码： class Stack(object): """ 栈 使用python列表实现 """

```
def __init__(self):  self.items = list()

def is_empty(self):  """判空"""  return self.items == []

def size(self):  """获取栈元素个数"""  return len(self.items)

def push(self, item):  """入栈"""  self.items.append(item)

def pop(self):  """出栈"""  self.items.pop()

def peek(self):  """获取栈顶元素"""  if self.is_empty():  raise IndexError("stack is empty")  return self.items[-1]
```

flask的jwt?

一篇文章需求分析：(Flask + flask-jwt 实现基于Json Web Token的用户认证授权)

jwt是flask的一个第三方库：flask-jwt-------->可以实现基于Json Web Token的用户认证授权

使用 JWT 让你的 RESTful API 更安全 什么是 JWT ?

JWT 及时 JSON Web Token，它是基于 RFC 7519 所定义的一种在各个系统中传递紧凑和自包含的 JSON 数据形式。

紧凑（Compact）：由于传送的数据小，JWT 可以通过GET、POST 和 放在 HTTP 的 header 中，同时也是因为小也能传送的更快。

自包含（self-contained）: Payload 中能够包含用户的信息，避免数据库的查询。 JSON Web Token 由三部分组成使用 .

分割开： Header Payload Signature 一个 JWT 形式上类似于下面的样子：xxxxx.yyyy.zzzz

Header 一般由两个部分组成： alg typ alg 是是所使用的 hash 算法例如 HMAC SHA256 或 RSA，typ 是 Token 的类型自然就是 JWT。 { "alg": "HS256", "typ": "JWT" }

JSON Web Token 的工作流程：

在用户使用证书或者账号密码登入的时候一个 JSON Web Token 将会返回，同时可以把这个 JWT 存储在local storage、或者 cookie 中，用来替代传统的在服务器端创建一个 session 返回一个 cookie。

当用户想要使用受保护的路由时候，应该要在请求得时候带上 JWT ，一般的是在 header 的 Authorization 使用 Bearer 的形式，一个包含的 JWT 的请求头的 Authorization 如下：

Authorization: Bearer <token>

这是一中无状态的认证机制，用户的状态从来不会存在服务端，在访问受保护的路由时候回校验 HTTP header 中 Authorization 的 JWT，

同时 JWT 是会带上一些必要的信息，不需要多次的查询数据库。

这种无状态的操作可以充分的使用数据的 APIs，甚至是在下游服务上使用，这些 APIs 和哪服务器没有关系，

因此，由于没有 cookie 的存在，所以在不存在跨域（CORS, Cross-Origin Resource Sharing）的问题。

gil锁的局限性和打破方式?

局限性:

在Cpython解释器中，同一个进程下开启的多线程，同一时刻只能有一个线程执行，无法利用多核优势

GIL存在原因 CPython在执行多线程的时候并不是线程安全的，所以为了程序的稳定性，加一把全局解释锁，能够确保任何时候都只有一个Python线程执行。

GIL的弊端

GIL对计算密集型的程序会产生影响。因为计算密集型的程序，需要占用系统资源。GIL的存在，相当于始终在进行单线程运算，这样自然就慢了。

IO密集型影响不大的原因在于，IO，input/output，这两个词就表明程序的瓶颈在于输入所耗费的时间，线程大部分时间在等待，所以它们是多个一起等（多线程）还是单个等（单线程）无所谓的。

这就好比，你在公交站等公交时，你们排队等公交（单线程）还是沿着马路一字排开等（多线程）是无所谓的。公交车（即input，即输入的资源）没来，哪种方式都是瞎折腾。

解决方案(打破方式)

multiprocessing

multiprocessing是一个多进程模块，开多个进程，每个进程都带一个GIL，就相当于多线程来用了。

multiprocessing的弊端 多线程与多进程一个不同点在于：

多线程是共享内存的，即这些线程共用一个内存地址。好处在于便于线程间数据通信和数据同步。

多进程，各个进程地址之间是独立的内存地址。这样不存内存地址之间通信就麻烦了。

综上所述，如果是IO密集型且对数据通信有需求，使用python 的threading模块也是可以的。

解决方法： 1. 使用多进程执行，此将要面临解决共享数据的问题，多用queue或pipe解决； 2. 使用Python多线程load C的module执行。

from ctypes import * form threading import Thread

#加载动态库 lib = cdll.LoadLibrary("./libdeadloop.so")

#创建一个子线程，让其执行c语言编写的函数，此函数是一个死循环 t = Thread(target=lib.DeadLoop) t.start()

```
while True:    pass
```

网友博客理解：

IL是限制同一个进程中只有一个线程进入Python解释器。。。。。

而线程锁是由于在线程进行数据操作时保证数据操作的安全性(同一个进程中线程之间可以共用信息，如果同时对数据进行操作，则会出现公共数据错误)

其实线程锁完全可以替代GIL，但是Python的后续功能模块都是加在GIL基础上的，所以无法更改或去掉GIL,这就是Python语言最大的bug...只能用多进程或协程改善，或者直接用其他语言写这部分

追问

GIL本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。

保护不同的数据的安全，就应该加不同的锁。

每执行一个python程序，就是开启一个进程，在一个python的进程内，不仅有其主线程或者由该主线程开启的其他线程，还有解释器开启的垃圾回收等解释器级别的线程， 所有的线程都运行在这一个进程内，

所以：

1、所有数据都是共享的，这其中，代码作为一种数据也是被所有线程共享的（test.py的所有代码以及Cpython解释器的所有代码）

2、所有线程的任务，都需要将任务的代码当做参数传给解释器的代码去执行，即所有的线程要想运行自己的任务，首先需要解决的是能够访问到解释器的代码。

在python的原始解释器CPython中存在着GIL（Global Interpreter Lock，全局解释器锁），因此在解释执行python代码时，会产生互斥锁来限制线程对共享资源的访问，

直到解释器遇到I/O操作或者操作次数达到一定数目时才会释放GIL。 所以，虽然CPython的线程库直接封装了系统的原生线程，但CPython整体作为一个进程，同一时间只会有一个获得GIL的线程在跑，

其他线程则处于等待状态。 这就造成了即使在多核CPU中，多线程也只是做着分时切换而已。 不过muiltprocessing的出现，已经可以让多进程的python代码编写简化到了类似多线程的程度了

我对 GIL的理解：

解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁。

GIL锁开始运作主线程做操作主线程完成操作GIL锁释放资源 所以多线程共同操作共享资源的时候，有一个线程竞得了资源，它就被GIL锁保护起来，其他线程只能是在那里等着，

但是这个时候，线程的休眠唤醒，全部会消耗CPU资源，所以嘞，就会慢。 Python语言和GIL解释器锁没有关系，它是在实现Python解析器(CPython)时所引入的一个概念，

同样一段代码可以通过CPython，PyPy，Psyco等不同的Python执行环境来执行，然而因为CPython是大部分环境下默认的Python执行环境。所以在很多人的概念里CPython就是Python，

也就想当然的把GIL归结为Python语言的缺陷，所有GIL并不是python的特性，仅仅是因为历史原因在Cpython解释器中难以移除。

GIL保证同一时刻只有一个线程执行代码，每个线程在执行过程中都要先获取GIL

线程释放GIL锁的情况：

在IO操作等可能会引起阻塞的system call之前,可以暂时释放GIL,但在执行完毕后,必须重新获取GIL Python 3.x使用计时器（执行时间达到阈值后，当前线程释放GIL）或Python 2.x，tickets计数达到100

Python使用多进程是可以利用多核的CPU资源的。 多线程爬取比单线程性能有提升，因为遇到IO阻塞会自动释放GIL锁

GIL只对计算密集型的程序有作用，对IO密集型的程序并没有影响，因为遇到IO阻塞会自动释放GIL锁 当需要执行计算密集型的程序时，

可以选择：1.换解释器，2.扩展C语言，3.换多进程等方案

GIL（Global Interpreter Lock）：全局解释器锁，python解释器在执行python字节码的时候会锁住解释器，导致其它的线程不能使用解释器，从而多线程情况下CPU上不去。

lupa是一个python调用lua的第三方库（https://pypi.python.org/pypi/lupa），lua_code是一段纯CPU计算的lua代码片段，之后开启了3个线程，可以发现CPU利用率达到了300% 之前在写一些python程序的时候，

如果是cpu密集的常常会使用多进程的方式，但是这样会有一些缺点：

1. 进程间共享数据特别麻烦，虽然multiprocessing库提供了很多进程间共享数据的方法，但是这些方法最后自己会成为瓶颈

2. 编程复杂度比较高

3. 主进程和子进程必然需要通信，进程间数据隔离，所以数据需要内存拷贝，成本高
相应的python代码：

```
#python lupa load
import lupa   lua = lupa.LuaRuntime()
LIBS = [    "./scripts/foo.lua",  ]
llibs = {}
def get_file_name(filename):
    import os
    (_, tmp) = os.path.split(filename)
    (f_name, ext) = os.path.splittext(tmp)
    return f_name
def load_libs():
    global LIBS, llibs    for lib_p in LIBS:     f = open(lib_p, 'r')
    code_str = f.readlines()     filename = get_file_name(lib_p)
    llibs[filename] = lua.execute('\n'.join(code_str))
if __name__ == '__main__':
    load_libs()
    print llibs['foo'].sayhi()
    print llibs['foo'].callback(100, 200, 300, 400)
    --foo.lualibfoo = {}
    function libfoo.sayhi()
    return "hi from lupa"   end  function libfoo.callback(a, b, c, d)    return a * b + c - d   endreturn libfoo
```

这样做有几点好处：

1. python写框架，lua写回调，每次调用一遍load_libs就相当于一次热更新

2. lua代码本身特别简单，可以交给策划配置，与热更新结合效果更好

3. python多线程结合lua使用可以突破python GIL的限制，后面补充一个demo

线程安全： 线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。 不会出现数据不一致或者数据污染。

线程不安全：就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

celery在某一时刻突然执行2回? 为什么? 否 怎么解决?

Celery是一个用Python开发的异步的分布式任务调度模块

网友遇到的工作问题以及解决方案：

使用 Celery Once 来防止 Celery 重复执行同一个任务 在使用 Celery 的时候发现有的时候 Celery 会将同一个任务执行两遍，我遇到的情况是相同的任务在不同的 worker 中被分别执行，

并且时间只相差几毫秒。

这问题我一直以为是自己哪里处理的逻辑有问题，后来发现其他人 也有类似的问题，然后基本上出问题的都是使用 Redis 作为 Broker 的，而我这边一方面不想将 Redis 替换掉，就只能在 task 执行的时候加分布式锁了。

不过在 Celery 的 issue 中搜索了一下，有人使用 Redis 实现了分布式锁，然后也有人使用了 Celery Once。 大致看了一下 Celery Once ，发现非常符合现在的情况，就用了下。

Celery Once 也是利用 Redis 加锁来实现, Celery Once 在 Task 类基础上实现了 QueueOnce 类，该类提供了任务去重的功能，

所以在使用时，我们自己实现的方法需要将 QueueOnce 设置为 base @task(base=QueueOnce, once={'graceful': True})

后面的 once 参数表示，在遇到重复方法时的处理方式，默认 graceful 为 False，那样 Celery 会抛出 AlreadyQueued 异常，手动设置为 True，则静默处理。

另外如果要手动设置任务的 key，可以指定 keys 参数

@celery.task(base=QueueOnce, once={'keys': ['a']}) def slow_add(a, b): sleep(30) return a + b

celery 任务突然不执行是为什么?

问题：

别人推送很多消息给我，用 tornado 接收然后传到 celery 里面处理 celery 进程刚启动还是没问题，运行一天半天 突然里面的任务都不处理了 重新启动下 就能把之前接收到的推送 一个个继续处理。。。

看不出是什么问题。。 打算修改下配置的处理任务的超时时间，看看能不能解决这个问题。

目前解决方案：

在服务器加一个定时刷新，开启时间长，处理大量消息，出现任务超时，端口占用，任务没有放掉，导致后期任务无法执行，需要服务器定时任务重启

问题：

最近在写一个分布式微博爬虫，主要就是使用celery做的分布式任务调度。celery确实比较好用，但是也遇到一些问题，我遇到的问题主要集中在定时任务和任务路由这两个部分。

本文不会讲解celery的基本使用，如果需要看celery入门教程的话，请点击这里跳转。 celery worker -A app_name -l info必须推荐在项目的根目录运行而且,这里的app_name必须是项目中的Celery实例的完整引用路径*。

如果不在项目根目录运行，那么相关的调用也得切换到app同级目录下，这一点可以通过命令行进行佐证

celery的定时任务会有一定时间的延迟。比如，我规定模拟登陆新浪微博任务每隔10个小时执行一次，那么定时任务第一次执行就会在开启定时任务之后的10个小时后才会执行。

而我抓取微博需要马上执行，需要带上cookie，所以不能等那1个小时。这个没有一个比较好的解决方法，可以使用celery的crontab()来代替schdule做定时，它会在启动的时候就执行。

解决方案

我采用的方法是第一次手动执行该任务，然后再通过schedule执行。 celery的定时任务可能会让任务重复。定时器一定只能在一个节点启动，否则会造成任务重复。

另外，如果当前worker节点都停止了，而beat在之后才停止，那么下一次启动worker的时候，它还会执行上一次未完成的任务，可能会有重复。

由于抓取用户和抓取用户关注、粉丝的任务耗时和工作量不同，所以需要使用任务路由，将任务按比重合理分配到各个分布式节点上，这就需要使用到celery提供的task queue。

如果单独使用task queue还好，但是和定时任务一起使用，就可能出现问题。我遇到的问题就是定时任务压根就不执行！开始我的配置大概就是这样

```
app.conf.update(
    CELERY_TIMEZONE='Asia/Shanghai',
    CELERY_ENABLE_UTC=True,
    CELERY_ACCEPT_CONTENT=['json'],
```

```
CELERY_TASK_SERIALIZER='json',
CELERY_RESULT_SERIALIZER='json',
CELERYBEAT_SCHEDULE={
    'user_task': {    'task': 'tasks.user.excute_user_task',    'schedule': timedelta(minutes=3),    },
    'login_task': {    'task': 'tasks.login.excute_login_task',    'schedule': timedelta(hours=10),    },
    },
CELERY_QUEUES=(
    Queue('login_queue', exchange=Exchange('login', type='direct'), routing_key='for_login'),
    Queue('user_crawler', exchange=Exchange('user_info', type='direct'), routing_key='for_user_info'),
    Queue('fans_followers', exchange=Exchange('fans_followers', type='direct'), routing_key='for_fans_followers')  )  )
```

结果过了一天发现定时任务并没有执行，后来把task加上了一个option字段，指定了任务队列，就可以了，

比如 'user_task': { 'task': 'tasks.user.excute_user_task', 'schedule': timedelta(minutes=3), 'options': {'queue': 'fans_followers', 'routing_key': 'for_fans_follwers'} },

部分分布式节点一直出现Received task，但是却不执行其中的任务的情况。这种情况下重启worker节点一般就可以恢复。

但是最好查查原因。通过查看flower的失败任务信息，才发现是插入数据的时候有的异常未被处理。这一点严格说来并不是celery的bug，不过也很令人费解。

所以推荐在使用celery的时候配合使用flower做监控。

Redis是否保证事务的一致性?

网友遇到问题：

”redis设计之初是简单高效，所以说在事务操作时命令是不会出错的，出错的可能性就是程序的问题 “ 那这样的意思就是把锅抛给程序咯?

如果程序能保证百分之百不出错那么关系型数据库还要啥事务呢?

redis事务报错时仍会执行所有命令，这样怎么保证一致性呢?

或者说白了redis根本就不支持事务只是冠以事务的名号而已。以上纯属个人见解 又专业人士可以解释大家讨论。

网友理解：

1、Redis事务主要用于不间断执行多条命令，即是存在引发错误的命令。Redis先执行命令，命令执行成功后才会记录日志，所以出现错误时无法回滚。

支持完整的acid会让Redis变得复杂也可能导致性能较低。 此外，使用lua脚本也可以保证Redis不间断执行多条命令。

2、找到网上这个解释比较到位： 单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。

事务可以理解为一个打包的批量执行脚本，但批量指令并非原子化的操作，中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。

redis事务

1、基本概念

1）什么是redis的事务?

简单理解，可以认为redis事务是一些列redis命令的集合，并且有如下两个特点：

a）事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

b）事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

2）事务的性质ACID

一般来说，事务有四个性质称为ACID，分别是原子性，一致性，隔离性和持久性。

a）原子性atomicity：redis事务保证事务中的命令要么全部执行要不全部不执行。有些文章认为redis事务对于执行错误不回滚违背了原子性，是偏颇的。

b）一致性consistency：redis事务可以保证命令失败的情况下得以回滚，数据能恢复到没有执行之前的样子，是保证一致性的，除非redis进程意外终结。

c）隔离性Isolation：redis事务是严格遵守隔离性的，原因是redis是单进程单线程模式，可以保证命令执行过程中不会被其他客户端命令打断。

d）持久性Durability：redis事务是不保证持久性的，这是因为redis持久化策略中不管是RDB还是AOF都是异步执行的，不保证持久性是出于对性能的考虑。

3）redis事务的错误

使用事务时可能会遇上以下两种错误：

a）入队错误：事务在执行 EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），

或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）。

b）执行错误：命令可能在 EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。

注：第三种错误，redis进程终结，本文并没有讨论这种错误。

2、redis事务的用法

redis事务是通过MULTI，EXEC，DISCARD和WATCH四个原语实现的。

MULTI命令用于开启一个事务，它总是返回OK。

MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

另一方面，通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务。

Redis的持久化方式?

两种持久化方式：

RDB和AOF 深入了解RDB和AOF的作用原理，剩下的就是根据实际情况来制定合适的策略了，再复杂一点，也就是定制一个高可用的，数据安全的策略了。

在RDB方式下，你有两种选择，一种是手动执行持久化数据命令来让redis进行一次数据快照，另一种则是根据你所配置的配置文件 的 策略，达到策略的某些条件时来自动持久化数据。

而手动执行持久化命令，你依然有两种选择，那就是save命令和bgsave命令。 save操作在Redis主线程中工作，因此会阻塞其他请求操作，应该避免使用。

bgSave则是调用Fork,产生子进程，父进程继续处理请求。子进程将数据写入临时文件，并在写完后，替换原有的.rdb文件。

Fork发生时，父子进程内存共享，所以为了不影响子进程做数据快照，在这期间修改的数据，将会被复制一份，而不进共享内存。

所以说，RDB所持久化的数据，是Fork发生时的数据。在这样的条件下进行持久化数据，如果因为某些情况宕机，则会丢失一段时间的数据。

如果你的实际情况对数据丢失没那么敏感，丢失的也可以从传统数据库中获取或者说丢失部分也无所谓，那么你可以选择RDB持久化方式。

再谈一下配置文件的策略，实际上它和bgsave命令持久化原理是相同的。

AOF持久化方式：

配置文件中的appendonly修改为yes。开启AOF持久化后，你所执行的每一条指令，都会被记录到appendonly.aof文件中。

但事实上，并不会立即将命令写入到硬盘文件中，而是写入到硬盘缓存，在接下来的策略中，配置多久来从硬盘缓存写入到硬盘文件。

所以在一定程度一定条件下，还是会有数据丢失，不过你可以大大减少数据损失。 这里是配置AOF持久化的策略。

redis默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

而no则是不主动进行同步操作，是默认30s一次。当然always一定是效率最低的，个人认为everysec就够用了，数据安全性能又高。

Redis也允许我们同时使用两种方式，再重启redis后会从aof中恢复数据，因为aof比rdb数据损失小嘛。

深入理解Redis的两种持久化方式：RDB每次进行快照方式会重新记录整个数据集的所有信息。RDB在恢复数据时更快，可以最大化redis性能，子进程对父进程无任何性能影响。

AOF有序的记录了redis的命令操作。意外情况下数据丢失甚少。他不断地对aof文件添加操作日志记录，你可能会说，这样的文件得多么庞大呀。

是的，的确会变得庞大，但redis会有优化的策略，比如你对一个key1键的操作，set key1 001 ，set key1 002, set key1 003。那优化的结果就是将前两条去掉咯，

那具体优化的配置在配置文件中对应的是
https://images2015.cnblogs.com/blog/686162/201608/686162-20160809211516715-145676984.png

前者是指超过上一次aof重写aof文件大小的百分之多少，会再次优化，如果没有重写过，则以启动时为主。

后者是限制了允许重写的最小aof文件大小。bgrewriteaof命令是手动重写命令，会fork子进程，在临时文件中重建数据库状态，对原aof无任何影响，

当重建旧的状态后，也会把fork发生后的一段时间内的数据一并追加到临时文件，最后替换原有aof文件，新的命令继续向新的aof文件中追加。

Redis数据库简介：

Redis是一种高级key-value数据库。它跟memcached类似，不过数据可以持久化，而且支持的数据类型很丰富。

有字符串，链表，集 合和有序集合。支持在服务器端计算集合的并，交和补集(difference)等，还支持多种排序功能。

所以Redis也可以被看成是一个数据结构服务 器。 Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；

也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

Redis数据库简介：

默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，

当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化），

另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。

那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

二者的区别　RDB存在哪些优势呢?

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。

比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。

通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢?

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF的优势有哪些呢?

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。

事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。

而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。

然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。

因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

AOF的劣势有哪些呢?

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。

rdb这个就更有些 eventually consistent的意思了。

常用配置

RDB持久化配置

Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

AOF持久化配置

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no #从不同步。高效但是数据不会被持久化。

PTSD-追光者
关注 - 27
粉丝 - 2
+加关注

« 上一篇： python
» 下一篇： 网络编程,并发编程

posted @ 2019-04-28 20:23 PTSD-追光者 阅读(3532) 评论(0) 编辑 收藏