五一分享汇总

```
javascript:document.body.contentEditable='true';
document.designMode='on';void0
打开网页 F12
console
整个网页就可以随便编辑了
```

什么是SQL注入?

SQL注入（SQLi）是一种注入攻击，，可以执行恶意SQL语句。它通过将任意SQL代码插入数据库查询，使攻击者能够完全控制Web应用程序后面的数据库服务器。攻击者可以使用SQL注入漏洞绕过应用程序安全措施；可以绕过网页或Web应用程序的身份验证和授权，并检索整个SQL数据库的内容；还可以使用SQL注入来添加，修改和删除数据库中的记录

如何防止SQL注入攻击?

不要使用动态SQL

不要将敏感数据保留在纯文本中

限制数据库权限和特权

我来单独解释一波我的理解:

```
select * from 表名 where id=1 or 1=1
```

那么这条sql 就是查全表了。

sql 语句的预处理是可以解决sql 注入的。

where 条件的强制类型转换，也是可以的。
 憧憬
xss 攻击:

xss就是攻击者在web页面插入恶意的Script代码，当用户浏览该页之时，嵌入其中web里面的Script代码会被执行，从而达到恶意攻击用户的特殊目的。

比如: 我们常见的在 mysql 中数据 js 代码

但是: 大家以为 XSS就是弹窗，其实错了,弹窗只是测试XSS的存在性和使用性。

如何避免呢: 就是在输入的时候没有做严格的过滤，而在输出的时候，也没有进行检查，转义，替换等

所以防范的方法就是，不信任任何用户的输入，对每个用户的输入都做严格检查，过滤，在输出的时候，对某些特殊字符进行转义，替换等

以上，为什么说Mysql预处理可以防止SQL注入

简单点理解：prepareStatement会形成参数化的查询，例如：1select * from A where tablename.id = ?传入参数'1;select * from B'如果不经过prepareStatement，会形成下面语句：1select * from A where tablename.id = 1;select * from B这样等于两次执行，但如果经过预处理，会是这样：1select * from A where tablename.id = '1;select * from B''1;select * from B'只是一个参数，不会改变原来的语法

ABA 问题

CAS 并不是万能的，CAS 更新有 ABA 问题。即 T1 读取内存变量为 `A` ,T2 修改内存变量为 `B` ,T2 修改内存变量为 `A` ,这时 T1 再 CAS 操作 `A` 时是可行的。但实际上在 T1 第二次操作 `A` 时，已经被其他线程修改过了。举一个现实情况下的例子：

小明账户上有100元。现在小明取钱，小强汇钱，诈骗分子盗刷三个动作同时进行。

1. 小明取50元。
2. 诈骗分子盗刷50元。
3. 小强给小明汇款50元。

此时，银行交易系统出问题，每笔交易无法通过短信告知小明。ABA问题就是：

1. 小明验证账户上有100元后，取出50元。—— 账上有50元。
2. 小强不会验证小明账户的余额，直接汇款50元。—— 账上有100元。
3. 诈骗分子验证账户有100元后，取出50元。—— 账上有50元。

小强没有告诉小明自己汇钱，小明也没收到短信，那么小明就一直以为只有自己取款操作，最后损失了50元。

AtomicStampedReference

对于 ABA 问题，比较有效的方案是引入版本号，内存中的值每发生一次变化，版本号都 `+1` ；在进行CAS操作时，不仅比较内存中的值，也会比较版本号，只有当二者都没有变化时，CAS才能执行成功。`AtomicStampedReference` 便是使用版本号来解决ABA问题的。类似的还有 `AtomicMarkableReference` ， `AtomicStampedReference` 是使用 pair 的 `int stamp` 作为计数器使用， `AtomicMarkableReference` 的 pair 使用的是 `boolean mark` 。

(不定项选择题) Python中单下划线_foo与双下划线__foo与__foo__的成员，下列说法正确的是？

A _foo 不能直接用于'from module import *'

B __foo解析器用_classname__foo来代替这个名字，以区别和其他类相同的命名

C __foo__代表python里特殊方法专用的标识

答案：ABC

```
# 字典内部结构排序
rows = [
{'fname':'Brian', 'lname': 'Jones', 'uid': 1003},
{'fname':'David', 'lname': 'Beazley', 'uid': 1002},
{'fname':'John', 'lname': 'Cleese', 'uid': 1001},
{'fname':'Big', 'lname': 'Jones', 'uid': 1004}
]
from operator import itemgetter
rows_by_fname = sorted(rows, key=itemgetter("fname"))
rows_by_uid = sorted(rows, key=itemgetter("uid"))
print(rows_by_fname)
print(rows_by_uid)
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

两种方案

1> 对传入的参数进行编码转义

2> 使用Python的MySQLdb模块自带的方法

第一种方案其实在很多PHP的防注入方法里面都有，对特殊字符进行转义或者过滤。

第二种方案就是使用内部方法，类似于PHP里面的PDO，这里对上面的数据库类进行简单的修改即可。

修改后的代码

```
class Database:
 aurl = '127.0.0.1'
 user = 'root'
 password = 'root'
 db = 'testdb'
 charset = 'utf8'

 def __init__(self):
  self.connection = MySQLdb.connect(self.aurl, self.user, self.password, self.db, charset=self.charset)
  self.cursor = self.connection.cursor()

 def insert(self, query, params):
  try:
   self.cursor.execute(query, params)
   self.connection.commit()
  except Exception, e:
   print e
   self.connection.rollback()

 def query(self, query, params):
  cursor = self.connection.cursor(MySQLdb.cursors.DictCursor)
  cursor.execute(query, params)
  return cursor.fetchall()

 def __del__(self):
  self.connection.close()
```

这里 execute 执行的时候传入两个参数，第一个是参数化的sql语句，第二个是对应的实际的参数值，函数内部会对传入的参数值进行相应的处理防止sql注入，实际使用的方法如下

```
preUpdateSql = "UPDATE `article` SET title=%s,date=%s,mainbody=%s WHERE id=%s"
mysql.insert(preUpdateSql, [title, date, content, aid])
```

这样就可以防止sql注入，传入一个列表之后，MySQLdb模块内部会将列表序列化成一个元组，然后进行escape操作。

动态代理

当动态生成的代理类调用方法时，会触发 `invoke` 方法，在 `invoke` 方法中可以对被代理类的方法进行增强。

```
// 1. 首先实现一个InvocationHandler，方法调用会被转发到该类的invoke()方法。
class LogInvocationHandler implements InvocationHandler{
    ...
    private Hello hello;
    public LogInvocationHandler(Hello hello) {
        this.hello = hello;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwat
        if("sayHello".equals(method.getName())) {
            logger.info("You said: " + Arrays.toString(args));
        }
        return method.invoke(hello, args);
    }
}
// 2. 然后在需要使用Hello的时候，通过JDK动态代理获取Hello的代理对象。
Hello hello = (Hello)Proxy.newProxyInstance(
    getClass().getClassLoader(), // 1. 类加载器
    new Class<?>[] {Hello.class}, // 2. 代理需要实现的接口，可以有多个
    new LogInvocationHandler(new HelloImp()));// 3. 方法调用的实际处理者
System.out.println(hello.sayHello("I love you!"));
```

通过动态代理可以很明显的看到它的好处，在使用静态代理时，如果不同接口的某些类想使用代理模式来实现相同的功能，将要实现多个代理类，但在动态代理中，只需要一个代理类就好了。

除了省去了编写代理类的工作量，动态代理实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景中。

- 继承了Proxy类，实现了代理的接口，由于java不能多继承，这里已经继承了Proxy类了，不能再继承其他的类，所以JDK的动态代理不支持对实现类的代理，只支持接口的代理。
- 提供了一个使用InvocationHandler作为参数的构造方法。
- 生成静态代码块来初始化接口中方法的Method对象，以及Object类的equals、hashCode、toString方法

弊端

代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用Java动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。

动态代理与静态代理的区别

- Proxy类的代码被固定下来，不会因为业务的逐渐庞大而庞大；
- 代理对象是在程序运行时产生的，而不是编译期；
- 可以实现AOP编程，这是静态代理无法实现的；
- 解耦，如果用在web业务下，可以实现数据层和业务层的分离。
- 动态代理的优势就是实现无侵入式的代码扩展。
- 静态代理这个模式本身有个大问题，如果类方法数量越来越多的时候，代理类的代码量是十分庞大的。所以引入动态代理来解决此类问题

AtomicInteger

`AtomicInteger` 是 Java 中常见的原子类，每种基础类型都对应 `Atomic***`。`AtomicInteger` 中最重要的就属于原子更新操作，这里我们来分析下 `getAndAdd` 的实现。

```
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            //获取 value 的偏移量
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;

    public final int getAndAdd(int delta) {
        //调用 unsafe.getAndAddInt
        return unsafe.getAndAddInt(this, valueOffset, delta);
    }
```

`getAndAdd` 中直接调用 `unsafe.getAndAddInt`，原子更新的逻辑都在 `UnSafe` 类中：

```
public final int getAndAddInt(Object var1, long var2, int var4) {
  int var5;
  //循环 CAS
  do {
    //获取 volatile 字段值
    var5 = this.getIntVolatile(var1, var2);
  } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

  return var5;
}
```

`getAndAdd` 就是通过循环CAS，来执行原子更新的逻辑。

序列化

ProtoBuffer

`Protocol Buffers` 是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 `RPC` 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。

Protobuf 的优点

- **Protobuf 更小、更快、也更简单**。你可以定义自己的数据结构，然后使用代码生成器生成的代码来读写这个数据结构。你甚至可以在无需重新部署程序的情况下更新数据结构。只需使用 `Protobuf` 对数据结构进行一次描述，即可利用各种不同语言或从各种不同数据流中对你的结构化数据轻松读写。
- **“向后”兼容性好**，人们不必破坏已部署的、依靠“老”数据格式的程序就可以对数据结构进行升级。这样您的程序就可以不必担心因为消息结构的改变而造成的大规模的代码重构或者迁移的问题。因为添加新的消息中的 `field` 并不会引起已经发布的程序的任何改变。
- **Protobuf 语义更清晰**，无需类似 XML 解析器的东西（因为 `Protobuf` 编译器会将 `.proto` 文件编译生成对应的数据访问类以对 `Protobuf` 数据进行序列化、反序列化操作）。
- **Protobuf 的编程模式比较友好**，简单易学，同时它拥有良好的文档和示例，对于喜欢简单事物的人们而言，Protobuf 比其他的技术更加有吸引力。

Protobuf 的不足

由于文本并不适合用来描述数据结构，所以 `Protobuf` 也不适合用来对基于文本的标记文档（如 HTML）建模。另外，由于 XML 具有某种程度上的自解释性，它可以被人直接读取编辑，在这一点上 `Protobuf` 不行，它以二进制的方式存储，除非你有 `.proto` 定义，否则你没法直接读出 `Protobuf` 的任何内容。

```python
# 怎样找出一个序列中出现次数最多的元素呢?
words = [
'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
'my', 'eyes', "you're", 'under'
]
from collections import Counter
word_counts = Counter(words)
print(word_counts.most_common(3))
print(word_counts["look"])
# 如何增加计数
morewords = ['why','are','you','not','looking','in','my','eyes']
for word in morewords:
    word_counts[word] +=1

print(word_counts["eyes"])
a = Counter(words)
b = Counter(morewords)
print(a)
print(b)
print(a+b)

```

python中__new__()与__init__()的区别

1.首先用法不同

__new__()用于创建实例，所以该方法是在实例创建之前被调用，它是类级别的方法，是个静态方法；

__init__() 用于初始化实例，所以该方法是在实例对象创建后被调用，它是实例级别的方法，用于设置对象属性的一些初始值。

由此可知，__new__()在__init__() 之前被调用。如果__new__() 创建的是当前类的实例，会自动调用__init__()函数，通过return调用的__new__()的参数cls来保证是当前类实例，如果是其他类的类名，那么创建返回的是其他类实例，就不会调用当前类的__init__()函数。

2.其次传入参数不同：

__new__()至少有一个参数cls，代表当前类，此参数在实例化时由Python解释器自动识别；

__init__()至少有一个参数self,就是这个__new__()返回的实例，__init__()在__new__()的基础上完成一些初始化的操作。

3.返回值不同：

__new__()必须有返回值，返回实例对象；

__init__()不需要返回值。

憧憬：二分查找

```
#BinarySearch: find the location of the target number
def BinarySearch(nums:list,x:int) -> int:
    '''
        nums: Sorted array from smallest to largest
        x: Target number
    '''
    left,right = 0,len(nums)-1
    while left <= right:
        mid = (left+right)//2
        if nums[mid] == x:
            return mid
        if nums[mid] < x:
            left = mid+1
        else:
            right = mid-1
    return None
```

xss 攻击：

xss就是攻击者在web页面插入恶意的Script代码，当用户浏览该页之时，嵌入其中web里面的Script代码会被执行，从而达到恶意攻击用户的特殊目的。

比如：我们常见的在 mysql 中数据 js 代码

但是：大家以为 XSS就是弹窗，其实错了，弹窗只是测试XSS的存在性和使用性。

如何避免呢：就是在输入的时候没有做严格的过滤，而在输出的时候，也没有进行检查，转义，替换等

所以防范的方法就是，不信任任何用户的输入，对每个用户的输入都做严格检查，过滤，在输出的时候，对某些特殊字符进行转义，替换等

linux中ctrl+z和ctrl+c的区别
ctrl+c和ctrl+z都是中断命令,但是他们的作用却不一样.
ctrl+c是强制中断程序的执行,
而ctrl+z的是将任务中断,但是此任务并没有结束,他仍然在进程中他只是维持挂起的状态,用户可以使用fg/bg操作继续前台或后台的任务,fg命令重新启动前台被中断的任务,bg命令把被中断的任务放在后台执行.
例如:
当你vi一个文件是,如果需要用shell执行别的操作,但是你又不打算关闭vi,因为你得
存盘推出,你可以简单的按下ctrl+z,shell会将vi进程挂起~,当你结束了那个shell操作之后,你可以用fg命令继续vi你的文件.

from子句与多表连接

from子句 【格式】from <表名>

1、单表查询

当前数据库：from <表名>

非当前数据库：from <数据库名 . 表名>

2、多表连接

(1)交叉连接 <表1> cross join <表2>

(2)内连接

① 等值连接 from <表1> inner join <表2> on <表1.列1> = <表2.列2>

② 不等连接 from <表1> inner join <表2> on <表1.列1> {> | < | >= | <=} <表2.列2>

③ 自连接 指一张表与其自身进行连接，主要用于同一列的数据比较

(3)外连接

① 左外连接 from <表1> left join <表2> on <表1.列1> {> | < | >= | <=} <表2.列2>

② 右外连接 from <表1>right join<表2> on <表1.列1> {> | < | >= | <=} <表2.列2>

(4)自然连接 自动通过两表的相同列名进行连接 from <表1> natural join <表2>

RedLock

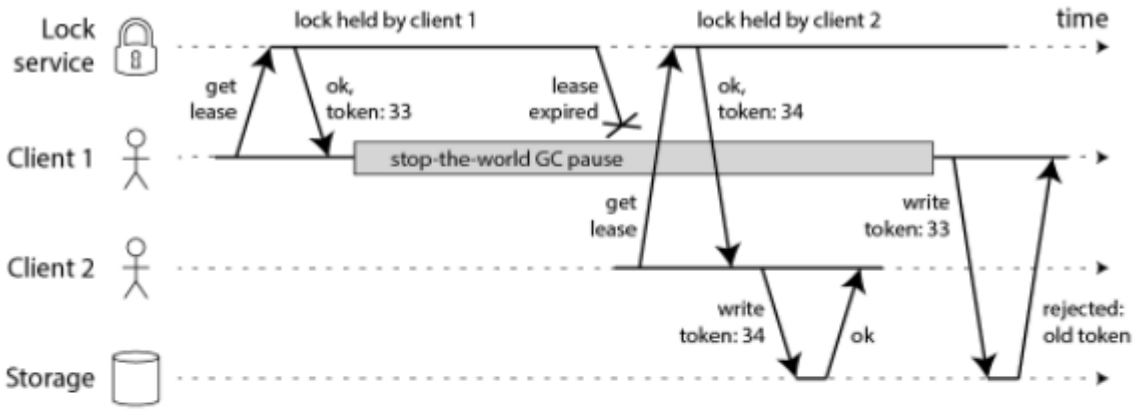
为了解决 Redis 单点的问题。 Redis 的作者提出了 RedLock 的解决方案。方案非常的巧妙和简洁。RedLock 的核心思想就是，**同时使用多个 Redis Master 来冗余，且这些节点都是完全的独立的，也不需要对这些节点之间的数据进行同步**。

假设我们有N个Redis节点，N应该是一个大于2的奇数。RedLock的实现步骤:

1. 取得当前时间
2. 使用单节点获取锁的方式，依次获取 N 个节点的 Redis 锁。
3. 如果获取到的锁的数量大于 $N/2+1$ 个，且获取的时间小于锁的有效时间(lock validity time)就认为获取到了一个有效的锁，锁自动释放时间就是最初的锁释放时间减去之前获取锁所消耗的时间。
4. 如果获取锁的数量小于 $N/2+1$ ，或者在锁的有效时间(lock validity time)内没有获取到足够的锁，就认为获取锁失败，这个时候需要向所有节点发送释放锁的消息。

对于释放锁的实现就很简单了，向所有的 Redis 节点发起释放的操作，无论之前是否获取锁成功。

缺陷



RedLock中，为了防止死锁，锁是具有过期时间的。

- 如果 Client 1 在持有锁的时候，发生了一次很长时间的 FGC 超过了锁的过期时间。锁就被释放了。
- 这个时候 Client 2 又获得了一把锁，提交数据。
- 这个时候 Client 1 从 FGC 中苏醒过来了，又一次提交数据。

这种情况下，数据就发生了错误。RedLock 只是保证了锁的高可用性，并没有保证锁的正确性。

1 pandas简介

1.1 前置课程numpy与scipy

numpy与scipy通常用于处理规范的数据，对于缺失值、数据的类型要求非常严格，然而实际情况下，原始数据通常不是很规范，且存在缺失值，或者数据类型混乱的情况，此时numpy和scipy将不再适用。

1.2 pandas的应用

Pandas最初被作为金融数据分析工具而开发出来，因此，pandas为时间序列分析提供了很好的支持。 Pandas的名称来自于面板数据（panel data）和python数据分析（data analysis）。panel data是经济学中关于多维数据集的一个术语，在Pandas中基本数据结构主要有Series和DataFrame两种，另外，也提供了 panel的数据类型。

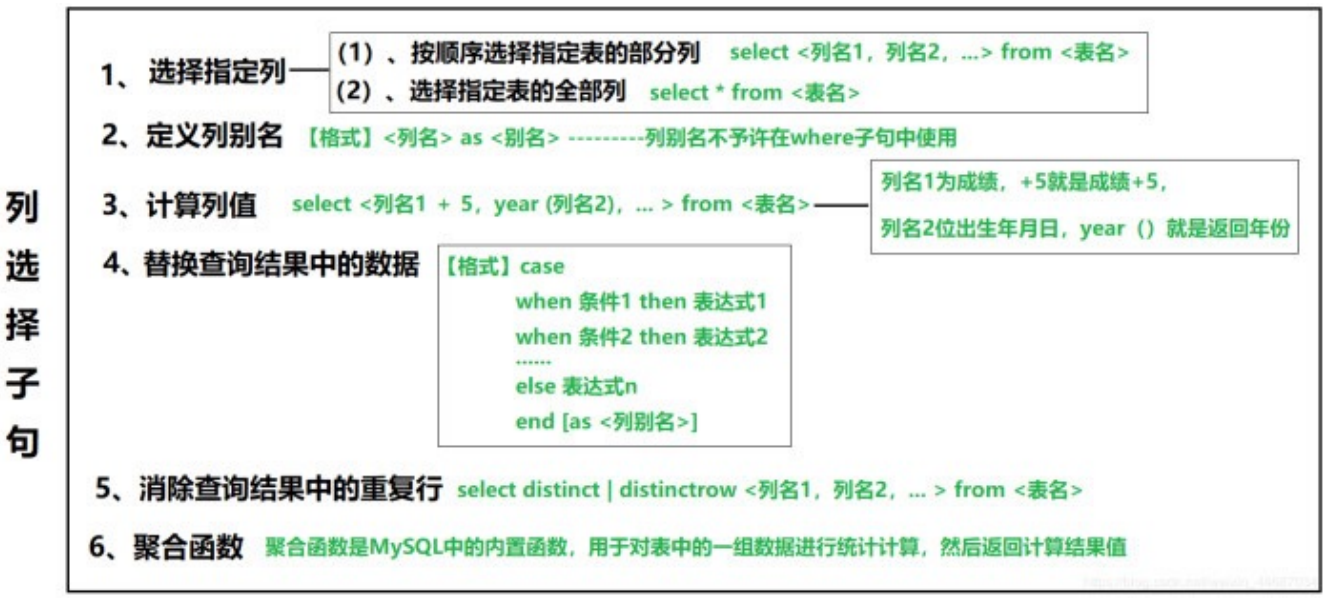
2 pandas基本数据结构 - Series

```
# 所有代码默认导入以下模块
from pandas import Series,DataFrame
import numpy as np
```

2.1 创建Series数据

2.1.1 通过列表创建

```
# 不指定索引，默认生成0，1，2索引
series1=Series(['小明','小红','小花'])
series1
# 通过index=[indeies]的方式指定索引
series2=Series(['小明','小红','小花'],index=['a','b','c'])
series2
>>  a    小明
    b    小红
    c    小花
    dtype: object
```



问题场景

在进行系统设计的过程中，首先问题场景的特点。秒杀系统是十分典型的高并发场景，其特点也十分显著：高并发、低库存、高瞬时流量。再者分析整个系统的输入输出，即大概的 API 网关拥有的功能：查（用户查询商品信息）、改（用户购买商品）。将系统的特点和功能分析完毕后，就可以根据这些信息进行系统设计。一个常规的秒杀系统从前到后，依次有：

```
前端页面 -> 代理服务 -> 后端服务 -> 数据库
```

根据这个流程，一般优化设计思路：将 **请求拦截在系统上游，降低下游压力**。在一个并发量大，实际需求小的系统中，应当尽量在前端拦截无效流量，降低下游服务器和数据库的压力，不然很可能造成数据库读写锁冲突，甚至导致死锁，最终请求超时。

整体优化手段包含：**缓存**、**限流**、**削峰（MQ）**、**异步处理**、**降级**、**熔断**、**SET化**、**快速扩容**

前端页面

- 资源静态化：将活动页面上的所有可以静态的元素全部静态化，尽量减少动态元素；通过CDN缓存静态资源，来抗峰值。
- 禁止重复提交：用户提交之后按钮置灰，禁止重复提交
- URL动态化：防止恶意抓取

代理服务

利用负载均衡（例如 Nginx 等）使用多个服务器并发处理请求，减小服务器压力。

后端服务

- 用户限流：在某一时间段内只允许用户提交一次请求，比如可以采取 IP 限流
- 业务拆分
- 利用 MQ 削峰
- 利用缓存应对大量查询请求
- 利用缓存应对写请求（注意数据一致性、持久性问题）：缓存也是可以应对写请求的，可把数据库中的库存数据转移到 Redis 缓存中，所有减库存操作都在 Redis 中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。

数据库

- 多数据库：防止数据热点问题
- 优化 SQL 防止死锁

【格式】select <列名1，列名2，...>------指明需要查询的列
from <表名>----------指明查询数据所涉及的表
[where <条件>]----------指明查询条件
[group by <分组依据>]----------将查询结果进行分组
[having <条件>]----------指明各分组应满足的条件，必须在group by后
[union]----------将查询结果组合到一起
[order by <排序依据>]----------将查询结果进行排序
[limit <行数>]----------指明查询数据的位置及行数
【功能】实现数据库中的数据查询

(不定项选择题) __new__和__init__的区别，说法正确的是？

A __new__是一个静态方法，而__init__是一个实例方法

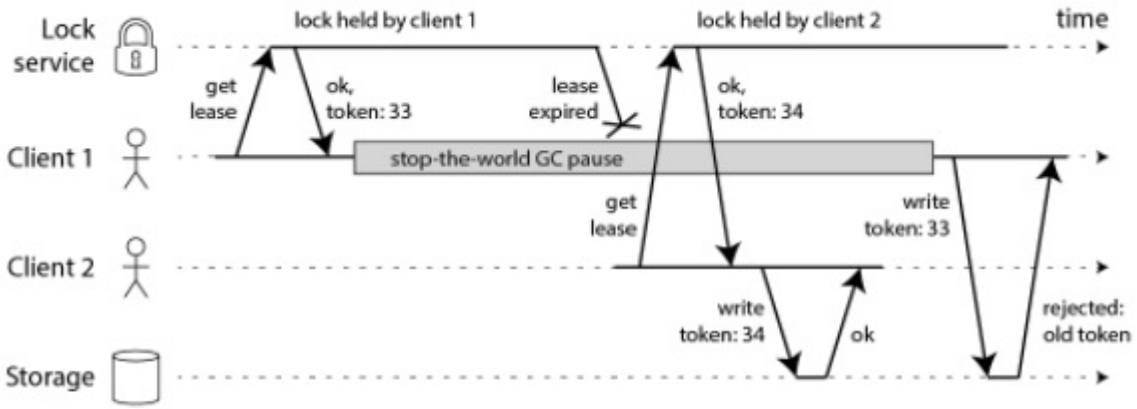
B __new__方法会返回一个创建的实例，而__init__什么都不返回

C 只有在__new__返回一个cls的实例时，后面的__init__才能被调用

D 当创建一个新实例时调用__new__，初始化一个实例时用__init__

答案：ABCD

解决方案可以为锁增加一个自增标识，类似于 Kafka 脑裂的处理方式：



同时 RedLock 是严重依赖系统时钟的一致性。如果某个 Redis Master的系统时间发生了错误，造成了它持有的锁提前过期被释放。

> 每一个系统设计都有自己的侧重或者局限。工程也不是完美的。在现实中工程中不存在完美的解决方案。我们应当深入了解其中的原理，了解解决方案的优缺点。明白选用方案的局限性。是否可以接受方案的局限带来的后果。架构本来就是一门平衡的艺术。