

基础+数据结构

1. 序列化和反序列化是什么，在代码中是如何实现的？

序列化：通过某种方式把数据结构或对象写入到磁盘文件中或通过网络传到其他节点的过程。例如 Django 中把模型类中的对象转化为 json 格式来存储。

反序列化：把磁盘对象或者把网络节点中传输的数据恢复为 python 的数据对象的过程。例如：把前端传来的 json 格式数据转换为 django 的模型类对象。

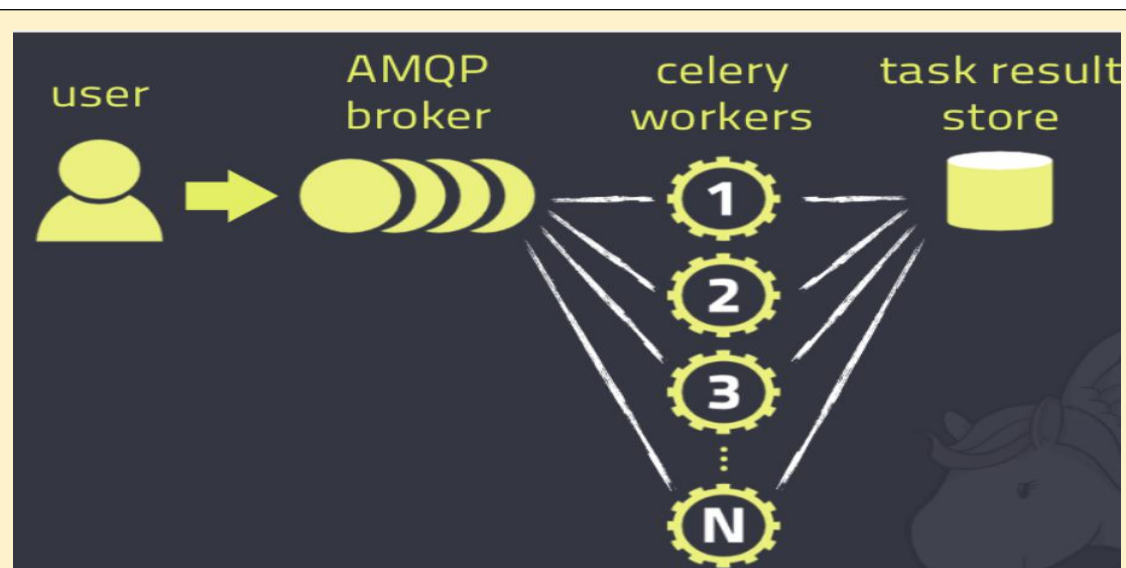
在 meiduo_mall 的商城项目中每个模块通过使用序列化器（serializers）来实现序列化或反序列化。例如下面的购物车序列化器，把 SKU 模型类中的字段通过 fields 来指明，然后序列化输出。也可以通过 ModelSerializer 自带的增加方法实现反序列化。

```
class CartSKUSerializer(serializers.ModelSerializer):
    """
    购物车商品数据序列化器
    """
    count = serializers.IntegerField(label='数量')

    class Meta:
        model = SKU
        fields = ('id', 'name', 'default_image_url', 'price', 'count')
```

2. Celery 的底层原理（透彻一些）？

celery 是基于 **python** 实现的一个异步任务的调度工具，同时还是一个任务队列，主要用于处理耗时的任务。架构如下：



celery 架构由三个模块组成：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 (Broker)：消息中间人，是任务调度队列，是一个独立的服务，是一个生产者消费之模式，生产者把任务放入队列中，消费者（worker）从任务队列中取出任务执行，任务的执行可以按照顺序依次执行也可以按照计划时间进行。但是 Broker 本身不提供队列服务，所以要集成第三方队列，推荐使用 RabbitMQ 或 Redis。

任务执行单元 (worker)：即执行任务的程序，可以有多个并发。它实时监控消息队列，获取队列中调度的任务，并执行它。

任务执行结果存储 (task result store)：由于任务的执行同主程序分开，如果主程序想获取任务执行的结果，就必须通过中间件存储。同消息中间人一样，存储也可以使用 RabbitMQ、Redis；另外，假如不需要保存执行的结果也可以不配置这个模块。

更多内容参考：

<http://docs.celeryproject.org/en/latest/getting-started/introduction.html>

3.对消息队列是怎么理解的？

从以下几个方面回答此问题：

什么是消息队列？

“消息队列”是在消息的传输过程中保存消息的容器。

“消息”是在两台计算机间传送的数据单位。消息可以非常简单，例如只包含文本字符串；也可以更复杂，可能包含嵌入对象。

消息被发送到队列中。“消息队列”是在消息的传输过程中保存消息的容器。

消息队列管理器在将消息从它的源中传到它的目标时充当中间人。队列的主要目的是提供路由并保证消息的传递；如果发送消息时接收者不可用，消息队列会保留消息，直到可以成功地传递它。

为什么需要消息队列？

主要原因是由于在高并发环境下，由于来不及同步处理，请求往往会发生堵塞，比如说，大量的 **insert**，**update** 之类的请求同时到达 **MySQL**，直接导致无数的行锁表锁，甚至最后请求会堆积过多，从而触发 **too many connections** 错误。通过使用消息队列，我们可以异步处理请求，从而缓解系统的压力。

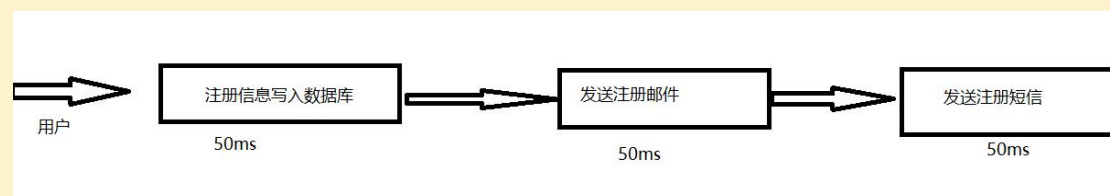
总结：消息队列中间件是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削锋等问题。实现高性能，高可用，可伸缩和最终一致性架构。是大型分布式系统不可缺少的中间件。目前在生产环境，使用较多的消息队列有 **ActiveMQ**，**RabbitMQ**，**ZeroMQ**，**Kafka**，**MetaMQ**，**RocketMQ** 等

应用的场景：

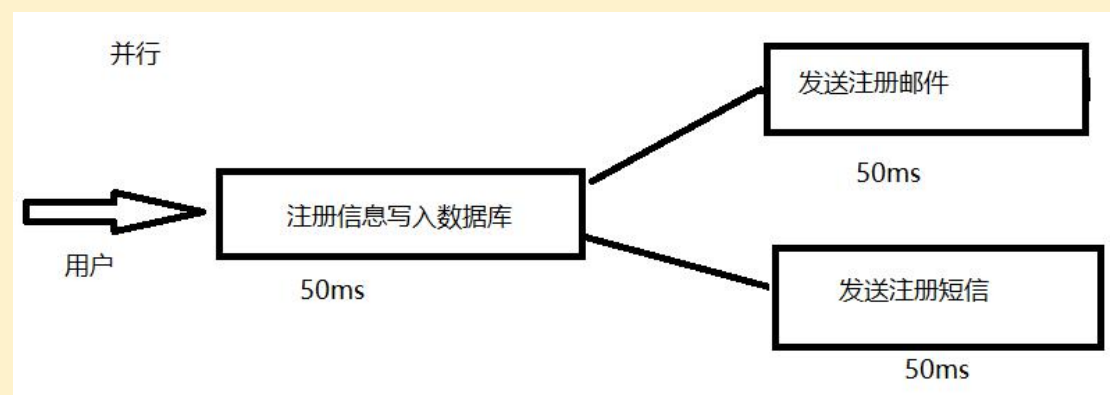
常用的使用场景：异步处理，应用解耦，流量削锋和消息通讯等业务场景。

以**异步处理应用场景**为例：用户注册后，需要发注册邮件和注册短信，传统的做法由两种：串行方式、并行方式。

(1) 串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。



(2) 并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间。

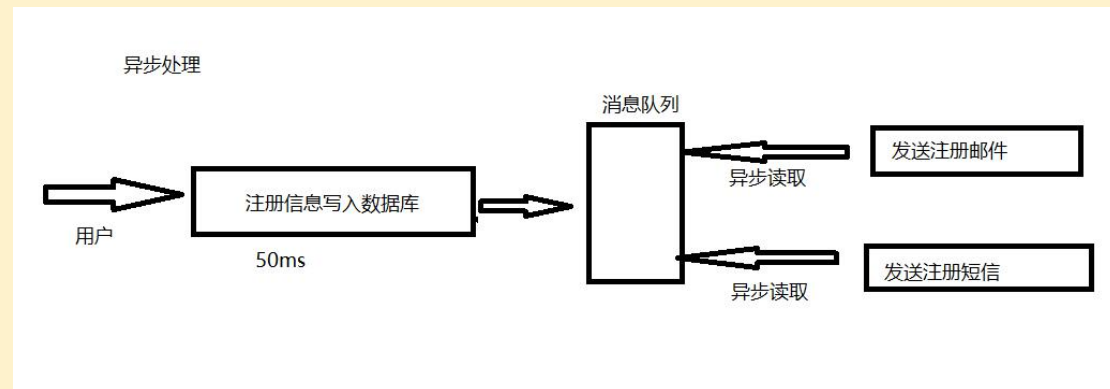


假设三个业务节点每个使用 **50** 毫秒种，不考虑网络等其他开销，则串行方式的时间是 **150** 毫秒，并行的时间可能是 **100** 毫秒。

因为 **CPU** 在单位时间内处理的请求数是一定的，假设 **CPU** 1 秒内吞吐量是 **100** 次。则串行方式 1 秒内 **CPU** 可处理的请求量是 **7** 次 ($1000/150$)。并行方式处理的请求量是 **10** 次 ($1000/100$)。

如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

使用消息队列，把不是必须的业务逻辑，异步处理。



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是 **50** 毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是 **50** 毫秒。因此架构改变后，系统的吞吐量提高到每秒 **20 QPS**。比串行提高了 **3** 倍，比并行提高了两倍。

消息队列的缺点：

系统可用性降低：系统引入的外部依赖越多，越容易挂掉，例如 **A** 只要调用 **BCD** 的接口就好了，但是加入 **MQ** 后，万一 **MQ** 挂掉，整个系统就不能使用了。

系统复杂性提高：硬生生加个 **MQ** 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？

一致性问题：**A** 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 **BCD** 三个系统那里，**BD** 两个系统写库成功了，结果 **C** 系统写库失败了，咋整？你这数据就不一致了

更多参考：

1. <https://blog.csdn.net/hxpjava1/article/details/81234929>
2. <https://www.cnblogs.com/xuyatao/p/6864109.html>
3. <https://blog.csdn.net/u014801403/article/details/80308353>

4、多线程在 web 项目中的应用

多线程一般在使用在进行 I/O 操作时，基于这个结论，提供以下几个使用场景：

1. 比如一个业务逻辑需要并行的操作几个文件的读写，还得是同步执行，不能

异步执行，这时候就可以开启多线程来读写这几个文件

2. 视图中需要请求多个第三方接口，仍然也是要求同步的，不能异步，这时候也可以用多线程去并行请求多个第三方接口

3. 比如在订单系统中，订单提交后就要修改商品的库存、商品的销量等这样的操作。

服务器以及部署问题

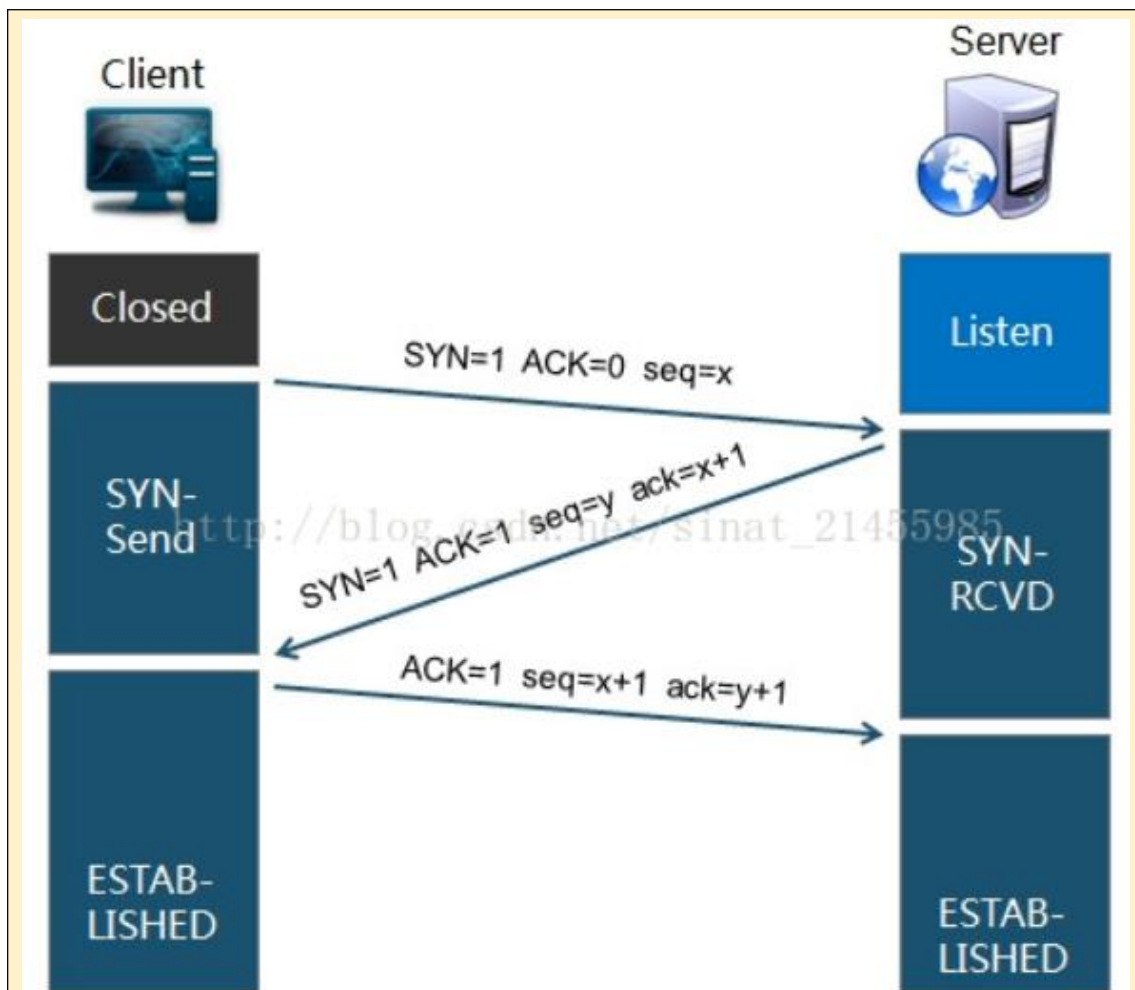
1. 说一下 **TCP** 的三次握手客户端在访问服务器的什么？在三次握手中服务器内部发生了什么事情？

建立起一个 **TCP** 连接需要经过“三次握手”：

1) **Client** 首先发送一个连接试探，**ACK=0** 表示确认号无效，**SYN = 1** 表示这是一个连接请求或连接接受报文，同时表示这个数据报不能携带数据，**seq = x** 表示 **Client** 自己的初始序号 (**seq = 0** 就代表这是第 0 号帧)，这时候 **Client** 进入 **syn_sent** 状态，表示客户端等待服务器的回复

2) **Server** 监听到连接请求报文后，如同意建立连接，则向 **Client** 发送确认。**TCP** 报文首部中的 **SYN** 和 **ACK** 都置 1，**ack = x + 1** 表示期望收到对方下一个报文段的第一个数据字节序号是 **x+1**，同时表明 **x** 为止的所有数据都已正确收到 (**ack=1** 其实是 **ack=0+1**，也就是期望客户端的第 1 个帧)，**seq = y** 表示 **Server** 自己的初始序号 (**seq=0** 就代表这是服务器这边发出的第 0 号帧)。这时服务器进入 **syn_rcvd**，表示服务器已经收到 **Client** 的连接请求，等待 **client** 的确认。

3) **Client** 收到确认后还需再次发送确认，同时携带要发送给 **Server** 的数据。**ACK** 置 1 表示确认号 **ack = y + 1** 有效 (代表期望收到服务器的第 1 个帧)，**Client** 自己的序号 **seq = x + 1** (表示这就是我的第 1 个帧，相对于第 0 个帧来说的)，一旦收到 **Client** 的确认之后，这个 **TCP** 连接就进入 **Established** 状态，就可以发起 **http** 请求了。



注意:

1. 三次握手，并没有传递数据，建立连接后才进入到数据传输的状态。
2. **SYN** (**synchronous**) 是 **TCP/IP** 建立连接时使用的握手信号。在客户机和服务器之间建立正常的 **TCP** 网络连接时，客户机首先发出一个 **SYN** 消息，服务器使用 **SYN+ACK** 应答表示接收到了这个消息，最后客户机再以 **ACK** 消息响应。这样在客户机和服务器之间才能建立起可靠的 **TCP** 连接，数据才可以在客户机和服务器之间传递。

2.不同应用服务器， session 怎么共享？

主要可以考虑下面几个方法，每个方法都有优缺点，具体实施时根据业务选择:

1.通过数据库 **mysql** 共享 **session**

a.采用一台专门的 **mysql** 服务器来存储所有的 **session** 信息。

用户访问随机的 **web** 服务器时，会去这个专门的数据库服务器 **check** 一下 **session** 的情况，以达到 **session** 同步的目的。

缺点就是：依赖性太强，**mysql** 服务器无法工作，影响整个系统；

b.将存放 **session** 的数据表与业务的数据表放在同一个库。如果 **mysql** 做了主从，需要每一

个库都需要存在这个表，并且需要数据实时同步。

缺点：用数据库来同步 **session**，会加大数据库的负担，数据库本来就是容易产生瓶颈的地方，如果把 **session** 还放到数据库里面，无疑是雪上加霜。上面的二种方法，第一点方法较好，把放 **session** 的表独立开来，减轻了真正数据库的负担。但是 **session** 一般的查询频率较高，放在数据库中查询性能也不是很好，不推荐使用这种方式。

2.通过 cookie 共享 session

把用户访问页面产生的 **session** 放到 **cookie** 里面，就是以 **cookie** 为中转站。

当访问服务器 **A** 时，登录成功之后将产生的 **session** 信息存放在 **cookie** 中；当访问请求分配到服务器 **B** 时，服务器 **B** 先判断服务器有没有这个 **session**，如果没有，在去看看客户端的 **cookie** 里面有没有这个 **session**，如果 **cookie** 里面有，就把 **cookie** 里面的 **session** 同步到 web 服务器 **B**，这样就可以实现 **session** 的同步了。

缺点：**cookie** 的安全性不高，容易伪造、客户端禁止使用 **cookie** 等都可能造成无法共享 **session**。

3.通过服务器之间的数据同步 session

使用一台作为用户的登录服务器，当用户登录成功之后，会将 **session** 写到当前服务器上，我们通过脚本或者守护进程将 **session** 同步到其他服务器上，这时当用户跳转到其他服务器，**session** 一致，也就不用再次登录。

缺陷：速度慢，同步 **session** 有延迟性，可能导致跳转服务器之后，**session** 未同步。而且单向同步时，登录服务器宕机，整个系统都不能正常运行。

4.通过 NFS 共享 Session

选择一台公共的 **NFS** 服务器 (**Network File Server**) 做共享服务器，所有的 **Web** 服务器登陆的时候把 **session** 数据写到这台服务器上，那么所有的 **session** 数据其实都是保存在这台 **NFS** 服务器上的，不论用户访问那台 **Web** 服务器，都要来这台服务器获取 **session** 数据，那么就能够实现共享 **session** 数据了。

缺点：依赖性太强，如果 **NFS** 服务器 **down** 掉了，那么大家都无法工作了，当然，可以考虑多台 **NFS** 服务器同步的形式。

5.通过 memcache 同步 session

memcache 可以做分布式，如果没有这功能，他也不能用来做 **session** 同步。他可以把 **web** 服务器中的内存组合起来，成为一个“内存池”，不管是哪个服务器产生的 **session** 都可以放到这个“内存池”中，其他的都可以使用。

优点：以这种方式来同步 **session**，不会加大数据库的负担，并且安全性比用 **cookie** 大大的提高，把 **session** 放到内存里面，比从文件中读取要快很多。

缺点：**memcache** 把内存分成很多种规格的存储块，有块就有大小，这种方式也就决定了，**memcache** 不能完全利用内存，会产生内存碎片，如果存储块不足，还会产生内存溢出。

6.通过 redis 共享 session

redis 与 **memcache** 一样，都是将数据放在内存中。区别的是 **redis** 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 **master-slave**(主从)同步。

根据实际开发应用，一般选择使用 **memcache** 或 **redis** 方式来共享 **session**。

3.在项目中用到的服务器数量，web 项目服务器用了几台，爬虫的服务器用了几台，本地的还是云服务器（为什么用云服务或者本地服务器，安全性？异地灾备？）

提示：

这是开放题、没有标准答案，以下仅供参考，要根据每个人说的集体情况改变。

在 Django 项目中最少用到以下服务器：

Nginx 提供反向代理和静态服务器的功能

Django 的业务服务器、

Mysql 的读写分离 2 台、reids 一台

celery 异步任务处理 Broker 和 Worker 台 2

FastDFS 进行图片存储 2 台

爬虫项目使用 2 台服务器。

有本地也有云服务器。例如像商品的数据都是在本地存储，而商品图片写入云服务器。

使用云服务器原因：

1. 云服务器费用少、并且使用简单，灵活，只要能联网就能使用，而本地存储携带不方便；
2. 云服务集成很多功能能提高数据使用率，能快速备份数据而不用增加服务器的硬件

4.你们部署服务器是几台，并发量是多大；怎么进行模拟抢购的 同一时间请求量是多少； 怎么防止带刷（黄牛） 如果说部署两台服务器 不同的进程 怎么实现乐观锁

Django 项目用到 5 台服务器。部署在一台上面，因为用户量比较少。

模拟抢购主要解决 2 个问题：

1.高并发对数据库产生的压力

2.竞争状态下如何解决库存的正确减少（“超卖”问题）

对于第一个问题可以使用 redis 解决，避免对数据库的直接操作较少数据防护的查询压力。

对于“超卖”项目中使用的是“乐观锁”解决的。

防止黄牛代刷是个开放题目，下面提供几种思路：

1. 对于一个账号，一次发起多个请求。

在程序入口处，一个账号只允许接受 1 个请求，其他请求过滤。实现方案，可以通过 Redis 这种内存缓存服务，写入一个标志位（只允许 1 个请求写成功，结合 watch 的乐观锁的特性），成功写入的则可以继续参加。

2. 对于账号一次发送多个请求

可以检测机器的 **ip 发送请求的频率**，假如某个固定 ip 的频率特别高，就弹出验证码来减少请求的频率。

乐观锁的实现原理：

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。

实现方式：可以在数据表中添加一个冗余字段，比如时间戳，在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则 OK，否则就是版本冲突。

5.高并发实际应用的经验，分布式实际经验

高并发：系统同时并行处理的请求数量。

web 网站都是追求在同时尽可能多的处理请求，我们可以从以下几个方面考虑尽可能提高网站的并发量：

1.减少数据库访问次数，

2. 文件和数据库分离（比如页面静态化），

3. 大数据分布式存储（比如：**Mysql** 的主从配置读写分离），

4. 服务器的集群负载均衡，

5. 页面缓存的使用（对于页面使用局部页面缓存或数据缓存），

6. **nosql** 内存数据库代替关系型数据库等

高并发项目中的例子：

1. 省市区的三级联动的数据，这些我们经常使用却不经常变化的数据，我们一般是一次全部读取出来缓存到 **web** 服务器的本地进行保存，假如有更新就重新读取；

2. 对于首页商品经常变化的数据使用 **redis** 进行缓存，比如说广告，可以使用缓存把广告查询出来的数据进行缓存，对于首页展示广告的部位使用局部缓存。来提高页面响应的速度。

分布式的案例

1. Django 项目中 MySQL 主从配置读写分离

在 Django 项目中为了提高数据库的查询速度，我们使用主从配置来实现

MySQL 的读写分离。但是虽然我们这么做能提高一部分的性能，假如用户的请求依旧很多，还是有访问的瓶颈，所以建议使用基于内存的非关系型数据库，比如 Mongo 或 Redis。

2. 爬虫中的分布式爬虫这两个角度来说。

爬虫项目使用 scrapy-redis 来实现分布式，由于 scrapy 本身的调度器 (scheduler) 并不支持分布式，所以使用第三方插件 scrapy-redis 来实现。scrapy-redis 是把所有请求的 url 通过调度器存入 redis 的有序集合 (zset) 中同时基于 fingerprint (或者 BloomFilter) 来实现去重，而 spider 只要启动多个线程或进程从 redis 的集合中取 url 进行解析数据就可以。

6. redis 宕机如何解决？如果是项目上线的宕机呢？

在主从模式下宕机要分为区分来看：

1. slave 从 redis 宕机

在 Redis 中从库重新启动后会自动加入到主从架构中，自动完成同步数据；如果从数据库实现了持久化，只要重新假如到主从架构中会实现增量同步。

2. Master 宕机

假如主从都没数据持久化，此时千万不要立马重启服务，否则可能会造成数据丢失，正确的操作如下：

1. 在 slave 数据上执行 SLAVEOF ON ONE,来断开主从关系并把 slave 升级为主库

2. 此时重新启动主数据库，执行 SLAVEOF,把它设置为从库，自动备份数据。以上过程很容易配置错误，可以使用简单的方法：redis 的哨兵 (sentinel) 的功能。

哨兵 (sentinel) 的原理：

Redis 提供了 sentinel (哨兵) 机制通过 sentinel 模式启动 redis 后，自动监控 master/slave 的运行状态，基本原理是：心跳机制+投票裁决。

每个 sentinel 会向其它 sentinel、master、slave 定时发送消息，以确认对方是否“活”着，如果发现对方在指定时间 (可配置) 内未回应，则暂时认为对方已挂 (所谓的“主观认为宕机” Subjective Down, 简称 SDOWN) 。

若“哨兵群”中的多数 sentinel，都报告某一 master 没响应，系统才认为该 master “彻底死亡”(即：客观上的真正 down 机，Objective Down, 简称 ODOWN)，通过一定的 vote 算法，从剩下的 slave 节点中，选一台提升为 master，然后自动修改相关配置。

补充：

哨兵的配置：

1. 复制 redis 中 sentinel.conf，根据情况进行配置

```
# 当前Sentinel服务运行的端口
port 26379

# Sentinel服务运行时使用的临时文件夹
dir /tmp

# sentinel连接的主redis, 此redis连接端口是192.168.1.103 6379
# 将此redis判断为失效至少需要1个 Sentinel进程的同意, 只要同意Sentinel的数量不达标, 自动failover就不会执行
sentinel monitor master 192.168.1.103 6379 1

# Sentinel认为Redis实例已经失效所需的时间是3000ms, 当足够数量的sentinel把此redis标记为失效, 才会发生故障迁移
sentinel down-after-milliseconds master 30000

# 指定了在执行故障转移时, 最多可以有多少个从Redis实例在同步新的主实例, 在从Redis实例较多的情况下这个数字越小, 同步的时间越长
# 完成故障转移所需的时间就越长
sentinel parallel-syncs master 1

# 如果未在18000ms内完成故障迁移, 就默认为失败
sentinel failover-timeout master 180000

# redis的sentinel的日志
logfile "/var/log/sentinel_log.log"
```

可以使用哨兵搭建高可用服务器, 哨兵模式还提供其他功能, 比如监控、通知、为客户端提供配置等。

关于使用哨兵搭建高可用服务器, 可以参考

<https://blog.csdn.net/shouhuzhezhishen/article/details/69221517> 来实现, 在此不再细说。

7. 数据库存储过程及触发器, 具体存储原理; 项目中多线程同时操作某段代码怎么处理?

方法 1: 可以使用锁把这段代码锁住, 让多线程排队依次操作这段代码。但是这个方法效率比较低。

方法 2: 使用乐观锁, 对于上个线程处理后通知其他线程来执行这段代码

存储过程

存储过程简介

我们常用的操作数据库语言 SQL 语句在执行的时候需要要先编译, 然后执行, 而**存储过程 (Stored Procedure)** 是一组为了完成特定功能的 SQL 语句集, 经编译后存储在数据库中, 用户通过指定存储过程的名字并给定参数 (如果该存储过程带有参数) 来调用执行它。

注意: 存储过程跟触发器有点类似, 都是一组 SQL 集, 但是存储过程是主动调用的, 且功能比触发器更加强大, 触发器是某件事触发后自动调用。

存储过程的特征:

1. 有输入输出参数, 可以声明变量, 有 **if/else, case, while** 等控制语句,

通过编写存储过程，可以实现复杂的逻辑功能；

2. 函数的普遍特性：模块化，封装，代码复用；

3. 速度快，只有首次执行需经过编译和优化步骤，后续被调用可以直接执行，省去以上步骤；

存储过程的使用场景：存储过程处理比较复杂的业务时比较实用。

复杂的业务逻辑需要多条 **SQL** 语句。这些语句要分别地从客户机发送到服务器，当客户机和服务器之间的操作很多时，将产生大量的网络传输。如果将这些操作放在一个存储过程中，那么客户机和服务器之间的网络传输就会大大减少，降低了网络负载。

存储过程的优点：

1. **增强了 sql 语言的功能和灵活性。**存储过程可以使用流控制语句进行编写，有很强的灵活性也可以完成复杂的计算。

2. **存储过程允许标准组件是编程。**存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的 SQL 语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

3. **存储过程能实现较快的执行速度。**如果某一操作包含大量的 Transaction-SQL 代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的 Transaction-SQL 语句在每次运行时都要进行编译和优化，速度相对要慢一些。

存储过程的缺点：

1. **消耗内存：**如果使用大量存储过程，那么使用这些存储过程的每个连接的内存使用量将会大大增加；此外，如果您在存储过程中过度使用大量逻辑操作，则 CPU 使用率也会增加，因为数据库服务器的设计不当于逻辑运算。

2. **难维护和难调试：**mysql 不支持存储过程的调试；存储过程学习难度大只是部分 DBA 的必备开发技能，对于普通的使用者维护难度加大。

3. **语法不支持跨数据库使用：**Mysql 和 Oracle 的存储过程的语法不通用，学习成本增加。

存储过程的使用：创建、调用

```
mysql>
mysql> 声明分隔符为$
mysql> delimiter $
mysql> drop procedure if exists hi; 如果存在存储过程hi就删除
-> create procedure hi() 创建新的存储过程
-> begin 开始的标识: begin
-> select * from tb_books;
-> end$ 结束的标识
Query OK, 0 rows affected, 1 warning (0.01 sec)

Query OK, 0 rows affected (0.01 sec) 执行的结果

mysql> █
```

解释：MySQL 默认以";"为分隔符，如果没有声明分割符，则编译器会把存储过程当成 SQL 语句进行处理，因此编译过程会报错，所以要事先用“DELIMITER

\$” 声明当前段分隔符，让编译器把两个"\$"之间的内容当做存储过程的代码，不会执行这些代码；“DELIMITER ;” 的意为把分隔符还原。

刚才创建存储过程 hi()时发现是函数，可以往里传入参数，参数共有三种：INT,OUT,INOUT.

- IN 参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值
- OUT:该值可在存储过程内部被改变，并可返回
- INOUT:调用时指定，并且可被改变和返回

以下案例仅供参考：

IN 案例

```
mysql> delimiter //
mysql> create procedure demo(IN p_in int)
-> begin
-> select p_in;
-> set p_in=2;
-> select p_in;
-> end //
Query OK, 0 rows affected (0.02 sec)

mysql> set @p_in=1 给p_in赋值
-> ;
-> ;
-> //
Query OK, 0 rows affected (0.01 sec)

mysql> call demo(@p_in); 调用demo同时传入参数
-> ;
-> //
+-----+
| p_in |
+-----+
| 1 |
+-----+
1 row in set (0.01 sec)

+-----+
| p_in |
+-----+
| 2 |
+-----+
1 row in set (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```

OUT 案例


```

mysql> delimiter //
mysql> create procedure demo_out(OUT p_out int)
-> begin
-> select p_out;
-> set p_out=2;
-> select p_out;
-> end;
-> //
Query OK, 0 rows affected (0.01 sec)

mysql> set @p_out=1;
-> //
Query OK, 0 rows affected (0.00 sec)

mysql> call demo_out(@p_out);
-> //
+-----+
| p_out |
+-----+
| NULL  |
+-----+
1 row in set (0.01 sec)

+-----+
| p_out |
+-----+
| 2     |
+-----+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> select @p_out;
-> //
+-----+
| @p_out |
+-----+
| 2     |
+-----+
1 row in set (0.00 sec)

```

ONOUT 例子

```

mysql>
mysql> delimiter //
mysql> create procedure demo_inout(INOUT p_inout int)
    -> begin
    -> select p_inout;
    -> set p_inout=2;
    -> select p_inout;
    -> end;
    -> //
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;
mysql> set @p_inout=1
    -> //
    -> ;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that c
ion for the right syntax to use near '/' at line 2
mysql>
mysql>
mysql>
mysql> set @p_inout=1;
Query OK, 0 rows affected (0.00 sec)

mysql> call demo_inout(@p_inout);
+-----+
| p_inout |
+-----+
|      1 |
+-----+
1 row in set (0.01 sec)

+-----+
| p_inout |
+-----+
|      2 |
+-----+
1 row in set (0.01 sec)

```

INOUT 参数 p_inout 类型 int

INOUT 参数可以在被调用时赋值或修改并且可以输出

修改

调用后发现修改后的也输出

参考资料:

1. <https://www.cnblogs.com/mark-chan/p/5384139.html>
2. <https://blog.csdn.net/xiaolyuh123/article/details/76147102>
3. 工具集 <http://tool.oschina.net/apidocs/apidoc?api=mysql-5.1-zh>

触发器

触发器简介:

触发器是与表关联的命名数据库对象，并在表发生特定事件时激活。触发器的一些用途是执行对要插入表中的值的检查，或者对更新中涉及的值执行计算。

触发器定义为在语句插入，更新或删除关联表中的行时激活。这些行操作是触发事件。例如，可以通过 **INSERT** 或 **LOAD DATA** 语句插入行，并为每个插入的行激活插入触发器。可以将触发器设置为在触发事件之前或之后激活。例如，您可以在插入表中的每一行之前或更新的每一行之后激活触发器。

简单的说，就是一张表发生了某件事（插入、删除、更新操作），然后自动触发了预先编写好的若干条 **SQL** 语句的执行

注意:

触发器是一种特殊类型的存储过程，它又不同于存储过程，
触发器主要是通过事件进行触发而被执行的，而存储过程可以通过存储过程名字而被直接调用

触发器的特征:

触发事件的操作和触发器里的 **SQL** 语句是一个事务操作，具有原子性,要么全部执行，要么都不执行。

触发器的使用场景: 满足特定条件触发响应的动作

触发器的优点:

比较适用于复杂的业务逻辑。比如，数据库中一条数据发生改变（更新、删除、增加）时，通过触发器让其他多张表发生改变。

触发器的缺点:

1. 增加维护的成本。有些业务逻辑在代码中处理，有些业务逻辑用触发器处理，会使后期维护变得困难
2. 学习成本增加: **Mysql** 和 **Oracle** 的触发器的语法格式是不一样的。

触发器的使用

语法:

Create trigger <触发器名称> Delimiter--触发器必须有名字，最多 64 个字符，分隔符结尾。

{ BEFORE | AFTER } --触发器有执行的时间设置：可以设置为事件发生前或后。

{ INSERT | UPDATE | DELETE } --同样也能设定触发的事件：它们可以在执行 **insert**、**update** 或 **delete** 的过程中触发。

ON <表名称> --触发器是属于某一个表的:当在这个表上执行插入、更新或删除操作的时候就导致触发器的激活。我们不能给同一张表的同一个事件安排两个触发器。

FOR EACH ROW --触发器的执行间隔: **FOR EACH ROW** 子句通知触发器 每隔一行执行一次动作，而不是对整个表执行一次。

<触发器 SQL 语句> --触发器包含所要触发的 **SQL** 语句：这里的语句可以是任何合法的语句，包括复合语句，但是这里的语句受的限制和函数的一样。

案例:

```
mysql>
mysql> create table my_goods(
  -> id int primary key auto_increment,
  -> name varchar(20) not null,
  -> inv int
  -> );
Query OK, 0 rows affected (0.04 sec)

mysql> create table my_orders(
  -> id int primary key auto_increment,
  -> goods_id int not null,
  -> goods_num int not null);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into my_goods values(null,'手机',1000),(null,'电脑',500),(null,'游戏机',100);
Query OK, 3 rows affected (0.04 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql>
```

创建触发器

```
mysql> delimiter $
mysql> create trigger after_order after insert on my_orders for each row
  -> begin
  -> update my_goods set inv = inv - new.goods_num where id=new.goods_id;
  -> end
  -> $
Query OK, 0 rows affected (0.02 sec)

mysql> delimiter ;
mysql>
```

插入一条订单数据

```
mysql>
mysql> select * from my_goods;
+----+-----+-----+
| id | name  | inv  |
+----+-----+-----+
| 1  | 手机  | 1000 |
| 2  | 电脑  | 500  |
| 3  | 游戏机| 100  |
+----+-----+-----+
3 rows in set (0.00 sec)

mysql> delimiter $
mysql> create trigger after_order after insert on my_orders for each row
  -> begin
  -> update my_goods set inv = inv - new.goods_num where id=new.goods_id;
  -> end
  -> $
Query OK, 0 rows affected (0.02 sec)

mysql> delimiter ;
mysql> insert into my_orders values(null,1,3);
Query OK, 1 row affected (0.02 sec)

mysql> select * from my_goods;
+----+-----+-----+
| id | name  | inv  |
+----+-----+-----+
| 1  | 手机  | 997  |
| 2  | 电脑  | 500  |
| 3  | 游戏机| 100  |
+----+-----+-----+
3 rows in set (0.00 sec)
```

查看触发器

```
mysql> show triggers \G: 查看触发器
***** 1. row *****
      Trigger: ins_sum
      Event: INSERT
      Table: account
      Statement: SET @sum = @sum + NEW.amount
      Timing: BEFORE
      Created: 2018-09-20 10:12:40.10
      sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
      Definer: root@%
      character_set_client: utf8
      collation_connection: utf8_general_ci
      Database Collation: utf8_general_ci
***** 2. row *****
      Trigger: after_order
      Event: INSERT
      Table: my_orders
      Statement: begin
update my_goods set inv = inv - new.goods_num where id=new.goods_id;
end
      Timing: AFTER
      Created: 2018-09-20 12:02:13.14
      sql_mode: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
      Definer: root@localhost
      character_set_client: utf8
      collation_connection: utf8_general_ci
      Database Collation: utf8_general_ci
rows in set: (0.01 sec)
```

删除触发器

```
mysql> drop trigger after_order; 删除触发器， drop trigger 触发器名字
Query OK, 0 rows affected (0.01 sec)

mysql> show triggers;
+-----+-----+-----+-----+-----+-----+-----+
| Trigger | Event | Table | Statement | Timing | Created | sql_mode |
+-----+-----+-----+-----+-----+-----+-----+
| after_order | INSERT | my_orders | begin  
update my_goods set inv = inv - new.goods_num where id=new.goods_id;  
end | AFTER | 2018-09-20 12:02:13.14 | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
| root@% | utf8 | utf8_general_ci | utf8_general_ci | |
+-----+-----+-----+-----+-----+-----+-----+
| ins_sum | INSERT | account | SET @sum = @sum + NEW.amount | BEFORE | 2018-09-20 10:12:40.10 | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
| root@% | utf8 | utf8_general_ci | utf8_general_ci | |
+-----+-----+-----+-----+-----+-----+-----+
rows in set: (0.01 sec)
```

更多参考:

1. <https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html>
2. <https://blog.csdn.net/yang1464657625/article/details/60463721>

数据库

1、Mysql 数据库中如何实现更快速的查找？

Mysql 实现快速查找可以从下面几个角度考虑实现, 具体方法有很多要配合使用:

1. 硬件和网络方面: 这要公司前期投入很多金钱。
2. 给相应的字段添加索引, 避免全文的扫描;
3. 缓存查询的结果, 减少数据库的查询次数;
4. 等价谓词的重写。比如 between and 更改为>=、<= 等, 查询的速度提升
5. 将外连接转化成内连接, 减少 IO 的操作。
6. 避免在 where 语句中使用! = 、<、>等操作符
7. 联接优化器, 使用不同连接顺序的表, 查到相同的结果。

参考:

1. <https://www.cnblogs.com/chiangchou/p/6158918.html>
2. <https://dev.mysql.com/doc/refman/8.0/en/performance-schema-queries.html>
3. <https://blog.csdn.net/u014421556/article/details/52063904>

2、Redis 有哪些特点？如何向 Redis 中存储数据？如何设置长连接？

Redis 特点：

1、速度快。Redis 是用 C 语言实现的，所有数据存储在内存中以键值对形式保存。

2、持久化

Redis 的所有数据存储在内存中，对数据的更新将异步地保存到磁盘上。

3、支持多种数据结构

Redis 支持五种数据结构：String、List、Set、Hash、Zset

4、支持多种编程语言。Java、php、Python、Ruby、Lua、Node.js

5、功能丰富。除了支持五种数据结构之外，还支持事务、流水线、发布/订阅、消息队列等功能。

6、源码简单，约 23000 行 C 语言源代码。

7、主从复制

主服务器（master）执行添加、修改、删除，从服务器执行查询。

8、高可用及分布式 支持高可用和分布式

如何存储数据，使用 python 操作数据库

```
4 def connection(host, port):
5
6     try:
7         # Redis()是StrictRedis的子类，
8         # connect = redis.Redis(host=host, port=port)
9         connect = redis.StrictRedis(host=host, port=port, db=1)
10        # 插入输入局，name是键， value是值
11        connect.set(name='hello', value='world')
12        # 输出
13        print connect.get(name='hello')
14
15    except :
16        print redis.RedisError
17
18    if __name__ == '__main__':
19        host = '0.0.0.0'
20        port = 6379
21        connection(host, port)
```

connection() > try

Run python2redis

/home/atguigu/.virtualenvs/py2/bin/python /home/atguigu/workspace/python2redis.py

world

Process finished with exit code 0

长连接概念:

所谓长连接，指在一个 TCP 连接上可以连续发送多个数据包，在 TCP 连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接。

短连接概念:

短连接是指通信双方有数据交互时，就建立一个 TCP 连接，数据发送完成后，则断开此 TCP 连接，即每次 TCP 连接只完成一对 CMPP 消息的发送。

设置长连接:

修改 redis.conf 的 tcp-keepalive=1，默认为 0，表示禁止长连接。

3、Redis 为什么比 MySQL 快？底层原理，不要说内存和磁盘的这种

1. Redis 存储的是 k-v 格式的数据。时间复杂度是 $O(1)$, 常数阶, 而 mysql 引擎的底层实现是 B+TREE, 时间复杂度是 $O(\log n)$ 是对数阶的。Redis 会比 Mysql 快一点。
2. Mysql 数据存储是存储在表中，查找数据时要先对表进行全局扫描或根据索引查找，这涉及到磁盘的查找，磁盘查找如果是条点查找可能会快点，但是顺序查找就比较慢。而 redis 不用这么麻烦，本身就是存储在内存中，会根据数据在内存的位置直接取出。
3. Redis 是单线程的多路复用 IO, 单线程避免了线程切换的开销，而多路复用 IO 避免了 IO 等待的开销，在多核处理器下提高处理器的使用效率可以对数据进行

分区，然后每个处理器处理不同的数据。

4、对 SQL 有哪些优化

SQL 优化一般理解是让 SQL 运行更快。实现 MySQL 优化可以从下面这些角度实现：

1. 在查询频率高的字段建立索引和缓存，对于经常查询的数据建立索引会提升查询速度，同时把经常用的数据进行缓存，避免对数据库的多次查询减少磁盘的 IO,节约时间。

2. 在 **where** 查询子语句上尽量避免对字段的 **NULL** 值判断，否则数据库引擎将会放弃索引而使用全表扫描。如：**select id from t where num is null**可以在 **num** 上设置默认值 **0**，确保表中 **num** 列没有 **null** 值，然后这样查询：**select id from t where num=0**

3.应尽量避免在 **where** 子句中使用**!=**或**<>**操作符，否则将引擎放弃使用索引而进行全表扫描。

4.应尽量避免在 **where** 子句中使用 **or** 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or num=20

可以这样查询：

select id from t where num=10

union all

select id from t where num=20

5.**in** 和 **not in** 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 **between** 就不要用 **in** 了：

select id from t where num between 1 and 3

6.下面的查询也将导致全表扫描：

select id from t where name like '%abc%'

7.应尽量避免在 **where** 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2=100

应改为：

select id from t where num=100*2

8.应尽量避免在 **where** 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3)='abc'--name 以 abc 开头的 id

应改为:

select id from t where name like 'abc%'

9.不要在 **where** 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

10.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

11.不要写一些没有意义的查询，如需要生成一个空表结构:

select col1,col2 into #t from t where 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样:

create table #t(...)

12.很多时候用 **exists** 代替 **in** 是一个好的选择:

select num from a where num in(select num from b)

用下面的语句替换:

select num from a where exists(select 1 from b where num=a.num)

13.并不是所有索引对查询都有效，**SQL** 是根据表中数据来进行查询优化的，当索引列有大量数据重复时，**SQL** 查询可能不会去利用索引，如一表中有字段 **sex**，**male**、**female** 几乎各一半，那么即使在 **sex** 上建了索引也对查询效率起不了作用。

14.索引并不是越多越好，索引固然可以提高相应的 **select** 的效率，但同时也降低了 **insert** 及 **update** 的效率，因为 **insert** 或 **update** 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

15.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

16.尽可能的使用 **varchar** 代替 **char**，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

17.任何地方都不要使用 **select * from t**，用具体的字段列表代替“*”，不要返回用不到的任何字段。

18.避免频繁创建和删除临时表，以减少系统表资源的消耗。

19.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

20.在新建临时表时，如果一次性插入数据量很大，那么可以使用 **select into** 代替 **create table**，避免造成大量 **log**，以提高速度；如果数据量不大，为了缓和系统表的资源，应先 **create table**，然后 **insert**。

21.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 **truncate table**，然后 **drop table**，这样可以避免系统表的较长时间锁定。

22.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考虑改写。

23.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

24.与临时表一样，游标并不是不可使用。对小型数据集使用 **FAST_FORWARD** 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。

在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

25.尽量避免大事务操作，提高系统并发能力。26.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

其他的 SQL 优化参考：

1. <https://blog.csdn.net/z719725611/article/details/52922695>
2. <https://blog.csdn.net/heqinghua217/article/details/78600967>
3. <https://blog.csdn.net/wwzuizz/article/details/54602058>
4. <https://www.cnblogs.com/easypass/archive/2010/12/08/1900127.html>

5、工作中对数据库设计有什么经验

以下描述仅仅是个人经验总结，仅供参考！！

我认为数据库设计分为库和表的设计，所以从这两方面着手介绍。

库的设计

- 1、数据库名称要明确，可以加前缀或后缀的方式，使其看起来有业务含义，比如数据库名称可以为 **Business_DB**(业务数据库)。
- 2、在一个企业中，如果依赖很多产品，但是每个产品都使用同一套用户，那么应该将用户单独构建一个库，叫做企业用户中心。
- 3、不同类型的数据应该分开管理，例如，财务数据库，业务数据库等。
- 4、由于存储过程在不同的数据库中，支持方式不一样，因此不建议过多使用和使用复杂的存储过程。为数据库服务器降低压力，不要让数据库处理过多的业务逻辑，将业务逻辑处理放到应用程序中。

表设计

1. 数据库表命名，将业务和基础表区分，采用驼峰表示法等。
2. 数据不要物理删除，应该加一个标志位，以防用户后悔时，能够恢复。
3. 添加时间，有添加时间可以明确知道记录什么时候添加的
4. 修改时间，可以知道记录什么时候被修改了，一旦数据出现问题，可以根据修改时间来定位问题。比如某人在这个时间做了哪些事。
5. 基本表及其字段之间的关系，应尽量满足第三范式。但是，满足第三范式的数据库设计，往往不是最好的设计。为了提高数据库的运行效率，常常需要降低范式标准：适当增加冗余，达到以空间换时间的目的
6. 若两个实体之间存在多对多的关系，则应消除这种关系。消除的办法是，在两者之间增加第三个实体。这样，原来一个多对多的关系，现在变为两个一对多的关系。要将原来两个实体的属性合理地分配到三个实体中去
- 7.对于主键的设计：
 1. 不建议用多个字段做主键，单个表还可以，但是关联关系就会有问题，主键自增是高性能的。导入导出就有问题
 2. 一般情况下，如果有两个外键，不建议采用两个外键作为联合主键，另建一个字段作为主键。除非这条记录没有逻辑删除标志，且该表永远只有一条此联合主键的记录。
 3. 一般而言，一个实体不能既无主键又无外键。在 **E—R** 图中，处于叶子部位的实体，可以定义主键，也可以不定义主键(因为它无子孙)，但必须要有外键(因为它有父亲)。

6、sql 语句中 where 和 having 哪个执行更快

sql 语句的书写顺序和执行顺序时不一样的，而是按照下面的顺序来执行：

from--where--group by--having--select--order by。

from:需要从哪个数据表检索数据

where:过滤表中数据的条件

group by:如何将上面过滤出的数据分组

having:对上面已经分组的数据进行过滤的条件

select:查看结果集中的哪个列，或列的计算结果

order by :按照什么样的顺序来查看返回的数据

通过执行顺序发现 where 其实是比 having 先执行，也就是说 where 速度更快。

where 和 having 的区别:

“Where” 是一个约束声明，使用 Where 来约束来自数据库的数据，Where 是在结果返回之前起作用的，且 Where 中不能使用聚合函数(例如 Sum)。

“Having” 是一个过滤声明，是在查询返回结果集以后对查询结果进行的过滤操作，在 Having 中可以使用聚合函数。

7、mongodb 在使用过程中有什么问题

关于连接方面

1. mongodb 目标计算机积极拒绝无法连接

此问题很多情况下是出现在 window 安装 Mongo 时，出现的主要原因是配置没配好:

1. 要把 **Mongo.exe** 添加系统化经中
 2. 在 **Mongo.exe** 的目录创建 **data/db** 目录来存放 **mongo** 的数据
 3. 在 **Mongo.exe** 的目录创建 **log** 文件来存放 **mongo** 的日志。
- 这三个缺一不可，一定要注意路径!!

2. 出现 mongo.js 的错误，如下:

```
Mongodbsshell version: 3.4.6
connecting to: test
Mon Mar 3 23:45:09.491 Error: couldn't connect to server 127.0.0.1:27017 at src/mongo/shell/mongo.js:145
exception: connect failed
```

解决方法: 删除 **data\db** 文件下面的 **mongo.lock** 文件 重启服务 OK

使用方面

1. **bson size** 不能超过 **16MB** 的限制。单个文档的 **BSON size** 不能超过 **16MB**。**find** 查询有时会遇到 **16MB** 的限制，譬如使用 **\$in** 查询的时候，**in** 中的数组元素不能太多

2. **Mongo** 对于时间的存储使用的 **MongoDate** 时间是 **UTC** 的，要使用日期格式来存储。

```

db.col.insert({"date": new Date(), num: 1})
db.col.insert({"date": new Date().toLocaleString(), num: 2})

db.col.find()
{
  "_id" : ObjectId("539944b14a696442d95eaf08"),
  "date" : ISODate("2014-06-12T06:12:01.500Z"),
  "num" : 1
}
{
  "_id" : ObjectId("539944b14a696442d95eaf09"),
  "date" : "Thu Jun 12 14:12:01 2014",
  "num" : 2
}

```

默认的

字符串形式

3. 使用 **count()** 统计数量错误。 需要使用 **aggregate pipeline** 来得到正确统计结果

4. 从 **shell** 中更新/写入到文档的数字，会变为 **float** 类型

5. DB 中的 **namespace** 数量太多导致无法创建新的 **collection**,

错误提示：**error: hashtable namespace index max chain reached:1335**，如何解决呢？

修改 **nssize** 参数并重启 **Mongodb**，这新 **nssize** 只会对新加入的 **DB** 生效，对以前已经存在的 **DB** 不生效，如果你想对已经存在的 **DB** 采用新的 **nssize**，必须在加大 **nssize** 重启之后新建 **DB**，然后把旧 **DB** 的 **collection** 复制到新 **DB** 中。

6. **moveChunk** 因旧数据未删除而失败。

错误日志：**"moveChunk failed to engage TO-shard in the data transfer: can't accept new chunks because there are still 1 deletes from previous migration"**。

意思是说，当前正要去接受新 **chunk** 的 **shard** 正在删除上一次数据迁移出的数据，不能接受新 **Chunk**，于是本次迁移失败。这种 **log** 里显示的是 **warning**，但有时候会发现 **shard** 的删除持续了十几天都没完成，查看日志，可以发现同一个 **chunk** 的删除在不断重复执行，重启所有无法接受新 **chunk** 的 **shard** 可以解决这个问题

解决办法： 重启 **Mongodb**

7. **mongo** 对数据分页慢。

这是因为 **Mongo** 是把分页的数据加载到内存中，由于内存限制，会越来越慢，如何解决？

分页时不要使用 **SKIP** 来实现。应该使用查询条件+排序+限制返回记录的方法，即边查询，边排序，排序之后，抽取上一页中的最后一条记录，作为当前分

页的查询条件，从而避免了 **skip** 效率低下的问题。

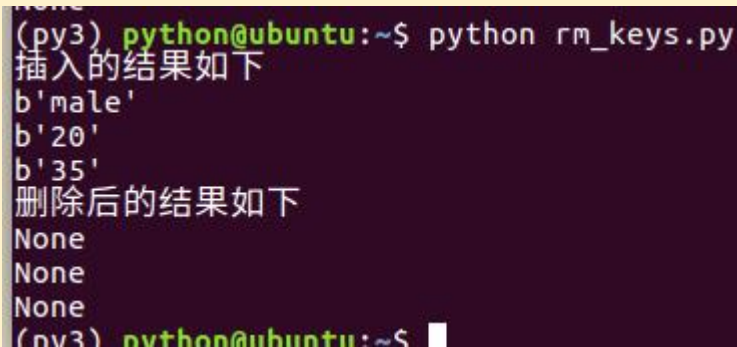
8、redis 中批量删除 key 写一个脚本

参考代码

```
import redis

conn = redis.Redis(host='localhost', port=6379, db=0)
#存数据
conn.set('tao_627', '35')
conn.set('age', '20')
conn.set('sex', 'male')
print('插入的结果如下')
print(conn.get('sex'))
print(conn.get('age'))
print(conn.get('tao_627'))
# 获取 keys, 返回 keys 的列表
keys = conn.keys()
conn.delete(*keys) # 使用*对列表解包
print('删除后的结果如下')
print(conn.get('sex'))
print(conn.get('age'))
print(conn.get('tao_627'))
```

执行结果:



```
(py3) python@ubuntu:~$ python rm_keys.py
插入的结果如下
b'male'
b'20'
b'35'
删除后的结果如下
None
None
None
(py3) python@ubuntu:~$
```

删除集群的 shell 脚本

1. 把待删除的 key 放入 txt 文件，比如下面例子:

```
1 key1
2 key2
3 key3
```

2. redis_delete_key.sh 中实现删除的脚本

```
1 redis_list=("127.0.0.1:6379" "127.0.0.1:6380")
2
3 for info in ${redis_list[@]}
4     do
5         echo "开始执行:$info"
6         ip=`echo $info | cut -d \: -f 1`
7         port=`echo $info | cut -d \: -f 2`
8         cat key.txt |xargs -t -n1 redis-cli -h $ip -p $port -c del
9     done
10    echo "完成"
```

存放待删除
key的文件
注意路径

执行结果

```
1 开始执行:127.0.0.1:6379
2 redis-cli -h 127.0.0.1 -p 6379 -c del key1
3 (integer) 0
4 redis-cli -h 127.0.0.1 -p 6379 -c del key2
5 (integer) 0
6 redis-cli -h 127.0.0.1 -p 6379 -c del key3
7 (integer) 0
8 开始执行:127.0.0.1:6380
9 redis-cli -h 127.0.0.1 -p 6380 -c del key1
10 (integer) 0
11 redis-cli -h 127.0.0.1 -p 6380 -c del key2
12 (integer) 0
13 redis-cli -h 127.0.0.1 -p 6380 -c del key3
14 (integer) 0
15 完成
```


9、redis 如何做持久化

持久化的概念：把数据放到断电也不会丢失的设备上。

Redis 实现持久化有两种方法：RDB 快照和 AOF

RDB 快照

RDB 快照

redis 是默认做数据持久化的，默认的方式是快照（**snapshotting**），把内存的数据写入本地的二进制文件 **dump.rdb** 文件中。

快照持久化实现原理：

Redis 借助了 **fork** 命令的 **copy on write** 机制。在生成快照时，将当前进程 **fork** 出一个子进程，然后在子进程中循环所有的数据，将数据写成为 **RDB** 文件，使用 **redis** 的 **save** 命令调用这个过程。

快照的配置有下面三个级别，配置是在 **redis** 的配置文件中。

#RDB方式的持久化是通过快照(snapshotting)完成的,当符合一定条件时Redis会自动将内存中的所有数据进行快照并存储在硬盘上。进行快照的条件可以由用户在配置文件中自定义,由两个参数构成:时间和改动的键的个数。当在指定的时间内被更改的键的个数大于指定的数值时就会进行快照。RDB是Redis默认采用的持久化方式,在配置文件中已经预置了3个条件:

save 900 1 # 900秒内有至少1个键被更改则进行快照

save 300 10 # 300秒内有至少10个键被更改则进行快照

save 60 10000 # 60秒内有至少10000个键被更改则进行快照

RDB 快照持久化的优点：

1. RDB 是一个非常紧凑的文件,它保存了某个时间点得数据集,非常适用于数据集的备份,比如您可以在每个小时报保存一下过去 24 小时内的数据,同时每天保存过去 30 天的数据,这样即使出了问题您也可以根据需求恢复到不同版本的数据集。
2. RDB 是一个紧凑的单一文件,很方便传送到另一个远端数据中心或者亚马逊的 S3（可能加密），非常适用于灾难恢复。
3. RDB 在保存 RDB 文件时父进程唯一需要做的就是 **fork** 出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他 IO 操作，所以 RDB 持久化方式可以最大化 **redis** 的性能。
4. 与 AOF 相比,在恢复大的数据集的时候，RDB 方式会更快一些。

RDB 快照持久化的缺点：

一旦数据库出现问题，那么我们的 RDB 文件中保存的数据并不是全新的，从上次 RDB 文件生成到 Redis 停机这段时间的数据全部丢掉了。

AOF

AOF 日志的全称是 **append only file**，从名字上我们就能看出来，它是一个追加写入的日志文件。

实现原理：

在使用 aof 时, redis 会将每一个收到的写命令都通过 write 函数追加到文件中, 当 redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容. Redis 还能对 AOF 文件通过 BGREWRITEAOF 重写, 使得 AOF 文件的体积不至于过大.

由于 AOF 是把操作日志写入日志文件中, 那么 AOF 的操作安全性如何保证? 可以在配置文件中通过配置告诉 redis 我们想要是哟个 fsync 函数强制写入到磁盘的时间.

配置分为三种:

1. appendfsync no

当设置 appendfsync 为 no 的时候, Redis 不会主动调用 fsync 去将 AOF 日志内容同步到磁盘, 所以这一切就完全依赖于操作系统的调试了. 对大多数 Linux 操作系统, 是每 30 秒进行一次 fsync, 将缓冲区中的数据写到磁盘上.

2. appendfsync everysec

当设置 appendfsync 为 everysec 的时候, Redis 会默认每隔一秒进行一次 fsync 调用, 将缓冲区中的数据写到磁盘. 但是当这一次的 fsync 调用时长超过 1 秒时. Redis 会采取延迟 fsync 的策略, 再等一秒钟. 也就是在两秒后再进行 fsync, 这一次的 fsync 就不管会执行多长时间都会进行. 这时候由于在 fsync 时文件描述符会被阻塞, 所以当前的写操作就会阻塞. 所以, 结论就是, 在绝大多数情况下, Redis 会每隔一秒进行一次 fsync. 在最坏的情况下, 两秒钟会进行一次 fsync 操作. 这一操作在大多数数据库系统中被称为 group commit, 就是组合多次写操作的数据, 一次性将日志写到磁盘.

3. appendfsync always

当设置 appendfsync 为 always 时, 每一次写操作都会调用一次 fsync, 这时数据是最安全的, 当然, 由于每次都会执行 fsync, 所以其性能也会受到影响.

AOF 的优点:

1. 使用 AOF 会更加持久化数据.
2. redis 可以在 aof 文件过大时使用 BGREWRITEAOF 进行重写
3. 保存的 aof 日志格式文件是按照 redis 协议的格式保存, 很容易读取.

AOF 的缺点:

1. 对于相同的数据集来说, AOF 文件的体积通常要大于 RDB 文件的体积
2. 速度没有 RDB 速度快.
3. 在写入内存数据的同时将操作命令保存到日志文件, 在一个并发更改上万的系统中, 命令日志是一个非常庞大的数据, 管理维护成本非常高, 恢复重建时间会非常长, 这样导致失去 aof 高可用性本意. 另外更重要的是 Redis 是一个内存数据结构模型, 所有的优势都是建立在对内存复杂数据结构高效的原子操作上, 这样就看出 aof 是一个非常不协调的部分. 其实 aof 目的主要是数据可靠性及高可用性.

10、乐观锁在进行更新数据的时候 数据库的查询命令用几条

乐观锁的实现

使用数据版本 (Version) 记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的“version”字段来实现。当读取数据时，将 version 字段的值一同读出，数据每更新一次，对此 version 值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的 version 值进行比对，如果数据库表当前版本号与第一次取出来的 version 值相等，则予以更新，否则认为是过期数据

所以乐观锁更新数据时，查询数据时执行一次。

11、既然是两条 SQL 语句 就必然会产生时间差 你们是怎么处理时间差的

框架

1、Django 与 Flask 相比的优缺点 为什么选用 Django

1. Django 相对 Flask 是比较“重大”的，因为 Django 中封装了很多现有的功能模块，比如后台管理网站、脚本管理工具 manage.py 等，而 Flask 并没有，要实现相同功能只能自己实现或要安装插件，比如安装 flask-script。
2. Django 是基于 MVT 架构模式，创建好项目后会自动帮我们实现代码模块的分割，不用像 flask 要自己配置

正是由于 Django 上述的特点才选用 Django 进行开发，能快速把搭建起来

业务问题（优先前面问题整理）

1、如何用最快的速度读出大小为 10G 的文件的行数？

```
with open('rm_keys.txt', 'r', encoding = 'utf-8') as f:  
    count = 0
```

```
# for line in f 将文件对象 f 视为一个可迭代的数据类型，会自动使用 IO 缓存和内存管理，这样就不必担心大文件了
```

```
for line in f:
    count += 1
    print(count)
```

2、图片管理为什么使用 **FastDFS**?为什么不用云端? 它的好处是什么?

FastDFS 比七牛云等云端存储会便宜一些。

FastDFS 是开源的轻量级分布式文件存储系统。它解决了大数据量存储和负载均衡等问题。特别适合以中小文件（建议范围：4KB < file_size < 500MB）为载体的在线服务。

优势：

1. 只能通过专用的 **API** 访问，不支持 **posix**，降低了系统的复杂度，处理效率高
2. 支持在线扩容，增强系统的可扩展性
3. 支持软 **RAID**，增强系统的并发处理能力及数据容错能力。**Storage** 是按照分组来存储文件，同组内的服务器上存储相同的文件，不同组存储不同的文件。**Storage-server** 之间不会互相通信。
4. 主备 **Tracker**，增强系统的可用性。
5. 支持主从文件，支持自定义扩展名
6. 文件存储不分块，上传的文件和 **os** 文件系统中的文件一一对应。
7. 相同内容的文件只存储一份，节约磁盘存储空间。对于上传相同的文件会使用散列方式处理文件内容，假如是一致就不会存储后上传的文件，只是把原来上传的文件在 **Storage** 中存储的 **id** 和 **ULR** 返回给客户端。

3、阿里云 **OSS** 上存储了什么东西?

阿里云对象存储服务 (**Object Storage Service**，简称 **OSS**) 为您提供基于网络的数据存取服务。使用 **OSS**，您可以通过网络随时存储和调用包括文本、图片、音频和视频等在内的各种非结构化数据文件。

阿里云 **OSS** 将数据文件以对象 (**object**) 的形式上传到存储空间 (**bucket**) 中。您可以进行以下操作：

1. 创建一个或者多个存储空间，向每个存储空间中添加一个或多个文件。
2. 通过获取已上传文件的地址进行文件的分享和下载。
3. 通过修改存储空间或文件的属性或元信息来设置相应的访问权限。
4. 在阿里云管理控制台执行基本和高级 **OSS** 任务。
5. 使用阿里云开发工具包或直接在应用程序中进行 **RESTful API** 调用执行基本和高级 **OSS** 任务。

参考文档: https://help.aliyun.com/document_detail/31883.html

4、有用过 **ElasticSearch** 吗? **Hystack** 如何对接 **ES**?

此处讲义上有讲解。

Haystack 为 **Django** 提供了模块化的搜索, 可以让你在不修改代码的情况下使用不同的搜索后端 (比如 **Solr**, **Elasticsearch**, **Whoosh**, **Xapian** 等等) 我们在 **django** 中可以通过使用 **haystack** 来调用 **Elasticsearch** 搜索引擎。

1) 安装

```
pip install drf-haystack
```

```
pip install elasticsearch==2.4.1
```

drf-haystack 是为了在 **REST framework** 中使用 **haystack** 而进行的封装 (如果在 **Django** 中使用 **haystack**, 则安装 **django-haystack** 即可) 。

2) 注册应用

```
INSTALLED_APPS = [
```

```
...
```

```
'haystack',
```

```
...
```

```
]
```

3) 配置

在配置文件中配置 **haystack** 使用的搜索引擎后端


```
# Haystack

HAYSTACK_CONNECTIONS = {

    'default': {

        'ENGINE':

'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',

        'URL': 'http://10.211.55.5:9200/', # 此处为 elasticsearch 运行的服务器 ip 地址,
端口号固定为 9200

        'INDEX_NAME': 'meiduo', # 指定 elasticsearch 建立的索引库的名称

    },
}


```

当添加、修改、删除数据时，自动生成索引

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

注意：

HAYSTACK_SIGNAL_PROCESSOR 的配置保证了在 Django 运行起来后，有新的数据产生时，haystack 仍然可以让 Elasticsearch 实时生成新数据的索引

4) 创建索引类

通过创建索引类，来指明让搜索引擎对哪些字段建立索引，也就是可以通过哪些字段的关键字来检索数据。

在 goods 应用中新建 search_indexes.py 文件，用于存放索引类

```
from haystack import indexes
```

```

from .models import SKU

class SKUIndex(indexes.SearchIndex, indexes.Indexable):
    """
    SKU 索引数据模型类
    """
    text = indexes.CharField(document=True, use_template=True)

    def get_model(self):
        """返回建立索引的模型类"""
        return SKU

    def index_queryset(self, using=None):
        """返回要建立索引的数据查询集"""
        return self.get_model().objects.filter(is_launched=True)

```

在 `SKUIndex` 建立的字段，都可以借助 `haystack` 由 `elasticsearch` 搜索引擎查询。

其中 `text` 字段我们声明为 `document=True`，表明该字段是主要进行关键字查询的字段，该字段的索引值可以由多个数据库模型类字段组成，具体由哪些模型类字段组成，我们用 `use_template=True` 表示后续通过模板来指明。

在 REST framework 中，索引类的字段会作为查询结果返回数据的来源。

6) 在 templates 目录中创建 text 字段使用的模板文件

具体在 templates/search/indexes/goods/sku_text.txt 文件中定义

```
{{ object.name }}
```

```
{{ object.caption }}
```

```
{{ object.id }}
```

此模板指明当将关键词通过 text 参数名传递时，可以通过 sku 的 name、caption、id 来进行关键字索引查询。

7) 手动生成初始索引

```
python manage.py rebuild_index
```

8) 创建序列化器

在 goods/serializers.py 中创建 haystack 序列化器

```
from drf_haystack.serializers import HaystackSerializer
```

```
class SKUSerializer(serializers.ModelSerializer):
```

```
    """
```

```
    SKU 序列化器
```

```
    """
```

```
    class Meta:
```

```
        model = SKU
```

```

        fields = ('id', 'name', 'price', 'default_image_url', 'comments')

class SKUIndexSerializer(HaystackSerializer):

    """
    SKU 索引结果数据序列化器
    """

    object = SKUSerializer(read_only=True)

class Meta:

    index_classes = [SKUIndex]

    fields = ('text', 'object')

```

说明：

下面的搜索视图使用 **SKUIndexSerializer** 序列化器用来检查前端传入的参数 **text**，并且检索出数据后再使用这个序列化器返回给前端；

SKUIndexSerializer 序列化器中的 **object** 字段是用来向前端返回数据时序列化的字段。

Haystack 通过 **Elasticsearch** 检索出匹配关键词的搜索结果后，还会在数据库中取出完整的数据库模型类对象，放到搜索结果的 **object** 属性中，并将结果通过 **SKUIndexSerializer** 序列化器进行序列化。所以我们可以声明搜索结果的 **object** 字段以 **SKUSerializer** 序列化的形式进行处理，明确要返回的搜索结果中每个数据对象包含哪些字段。

如，通过上面两个序列化器，最终的返回结果形式如下：

```
[
  {
    "text": "华为 HUAWEI P10 Plus 6GB+128GB 钻雕蓝 移动联通电信 4G 手机 双卡  
双待\nwifi 双天线设计！徕卡人像摄影！P10 徕卡双摄拍照，低至 2988 元！ \n11",
    "object": {
      "id": 11,
      "name": "华为 HUAWEI P10 Plus 6GB+128GB 钻雕蓝 移动联通电信 4G 手  
机 双卡双待",
      "price": "3788.00",
      "default_image_url":
"http://image.meiduo.site:8888/group1/M00/00/02/CtM3BVRdG6AYdapAAcPaeOq  
MpA1594598",
      "comments": 2
    }
  },
  {
    "text": "华为 HUAWEI P10 Plus 6GB+128GB 玫瑰金 移动联通电信 4G 手机 双卡  
双待\nwifi 双天线设计！徕卡人像摄影！P10 徕卡双摄拍照，低至 2988 元！ \n14",
    "object": {
      "id": 14,
      "name": "华为 HUAWEI P10 Plus 6GB+128GB 玫瑰金 移动联通电信 4G 手
```


机 双卡双待",

```
"price": "3788.00",
```

```
"default_image_url":
```

```
"http://image.meiduo.site:8888/group1/M00/00/02/CtM3BVRdMSAaDUtAAVslh9vkK04466364",
```

```
"comments": 1
```

```
}
```

```
}
```

```
]
```

9) 创建视图

在 `goods/views.py` 中创建视图

```
from drf_haystack.viewsets import HaystackViewSet
```

```
class SKUSearchViewSet(HaystackViewSet):
```

```
    """
```

```
    SKU 搜索
```

```
    """
```

```
    index_models = [SKU]
```

```
    serializer_class = SKUIndexSerializer
```

注意:

该视图会返回搜索结果的列表数据，所以如果可以为视图增加 REST framework 的分页功能。

我们在实现商品列表页面时已经定义了全局的分页配置，所以此搜索视图会使用全局的分页配置。

返回的数据举例如下：

```
{
  "count": 10,
  "next": "http://api.meiduo.site:8000/skus/search/?page=2&text=%E5%8D%8E",
  "previous": null,
  "results": [
    {
      "text": "华为 HUAWEI P10 Plus 6GB+64GB 钻雕金 移动联通电信 4G 手机  
双卡双待\nwifi 双天线设计！徕卡人像摄影！P10 徕卡双摄拍照，低至 2988 元！\n",
      "id": 9,
      "name": "华为 HUAWEI P10 Plus 6GB+64GB 钻雕金 移动联通电信 4G 手机  
双卡双待",
      "price": "3388.00",
      "default_image_url":
"http://10.211.55.5:8888/group1/M00/00/02/CtM3BVRcUeAHp9pAARfIK95am88523  
545",
      "comments": 0
    },
  ],
}
```

```
{
    "text": "华为 HUAWEI P10 Plus 6GB+128GB 钻雕金 移动联通电信 4G 手机  
双卡双待\nwifi 双天线设计! 徕卡人像摄影! P10 徕卡双摄拍照, 低至 2988 元! \n10",
    "id": 10,
    "name": "华为 HUAWEI P10 Plus 6GB+128GB 钻雕金 移动联通电信 4G 手  
机 双卡双待",
    "price": "3788.00",
    "default_image_url":
"http://10.211.55.5:8888/group1/M00/00/02/CtM3BVrRchWAMc8rAARfIK95am88158  
618",
    "comments": 5
}
]
```

10) 定义路由

通过 REST framework 的 router 来定义路由

```
router = DefaultRouter()
```

```
router.register('skus/search', views.SKUSearchViewSet, base_name='skus_search')
```

```
...
```

```
urlpatterns += router.urls
```

11) 测试

我们可以 GET 方法访问如下链接进行测试

<http://api.meiduo.site:8000/skus/search/?text=wifi>

5、高并发服务设计和实现 怎么做？

在项目我们使用下面技术实现高并发：

高并发项目中的例子：

1. 省市区的三级联动的数据，这些我们经常使用却不经常变化的数据，我们一般是一次全部读取出来缓存到 **web** 服务器的本地进行保存，假如有更新就重新读取；

2. 对于首页商品经常变化的数据使用 **redis** 进行缓存，比如说广告，可以使用缓存把广告查询出来的数据进行缓存，对于首页展示广告的部位使用局部缓存。来提高页面响应的速度。

6、购物车中添加数据的时候数据库数据减少吗？

没有，仅仅是把商品信息添加到 **cookie** 或 **redis** 中，没有对商品的数量进行更改

7、订单待支付时数据库中数据减少不？

在创建订单时数据库中的商品数据已经减少，在订单未支付时数据库中的商品数据是没变化，只是把此订单中商品的信息展示而已

8、如何设置（购物车或待支付账单）倒计时 30 分钟，数据库数据减少不？

这实现起来方法很多，下面仅提供思路：

1. 在创建订单时，添加额外字段，记录订单的创建时间。当用户进入此订单页面时就显示倒计时。前端获取当前时间，然后减去订单的创建时间，如果大于 30 分钟就取消订单，取消订单就修改商品的库存和销量；如果没有超时就继续倒计时。

2. 可以使用定时任务 **crontab**，在创建订单时就生成一个定时任务，设定为 30 分钟后用户未支付就取消订单，取消订单就把商品的库存和销量做相应的修改。

3. 使用 **celery** 异步任务实现。在生成订单时同时调用 **celery** 的 **countdown** 函数，只要传入 30 分钟的时间，时间到了订单未支付，就取消订单。

9、为什么使用 **CELERY** 而不使用线程发送耗时任务。

主要是因为并发比较大的时候，线程切换会有开销时间，假如使用线程池会限制并发的数量；同时多线程间的数据共享维护比较麻烦。

而 **celery** 是异步任务处理，是分布式的任务队列。它可以让任务的执行同主程序完全脱离，甚至不在同一台主机内。它通过队列来调度任务，不用担心并发量高时系统负载过大。它可以用来处理复杂系统性能问题，却又相当灵活易用。

10、token 和 jwt 存在什么区别

最直观的：**token** 需要查库验证 **token** 是否有效，而 **JWT** 不用查库或者少查库，直接在服务端进行校验，并且不用查库。

因为用户的信息及加密信息在第二部分 **payload** 和第三部分签证中已经生成，只要在服务端进行校验就行，并且校验也是 **JWT** 自己实现的。

例如下面这个例子：

现在有一个接口 **/vip** 只能是 **vip** 用户访问，我们看看服务端如何根据 **Token** 判断用户是否有效。

普通 **token** 版：

1. 查库判断是否过期
2. 查库判断时候是 **VIP**

JWT 版本：

假如 **payload** 部分如下:

```
{  
  "exp": 1518624000,  
  "isVip": true,  
  "uid":1  
}
```

1. 解析 JWT

2. 判断签名是否正确, 根据生成签名时使用的密钥和加密算法, 只要这一步过了就说明是 **payload** 是可信的

3. 判断 JWT token 是否过期, 根据 **exp**, 判断是否是 VIP, 根据 **isVip**

JWT 版是没有查库的, 他所需要的基础信息可以直接放到 JWT 里, 服务端只要判断签名是否正确就可以判断出该用户是否可以访问该接口, 当然 JWT 里的内容也不是无限多的, 其他更多的信息我们就可以通过 **id** 去查数据库

11、你的订单在支付时, 点击提交订单后库存已经更改, 长时间未支付怎么处理的

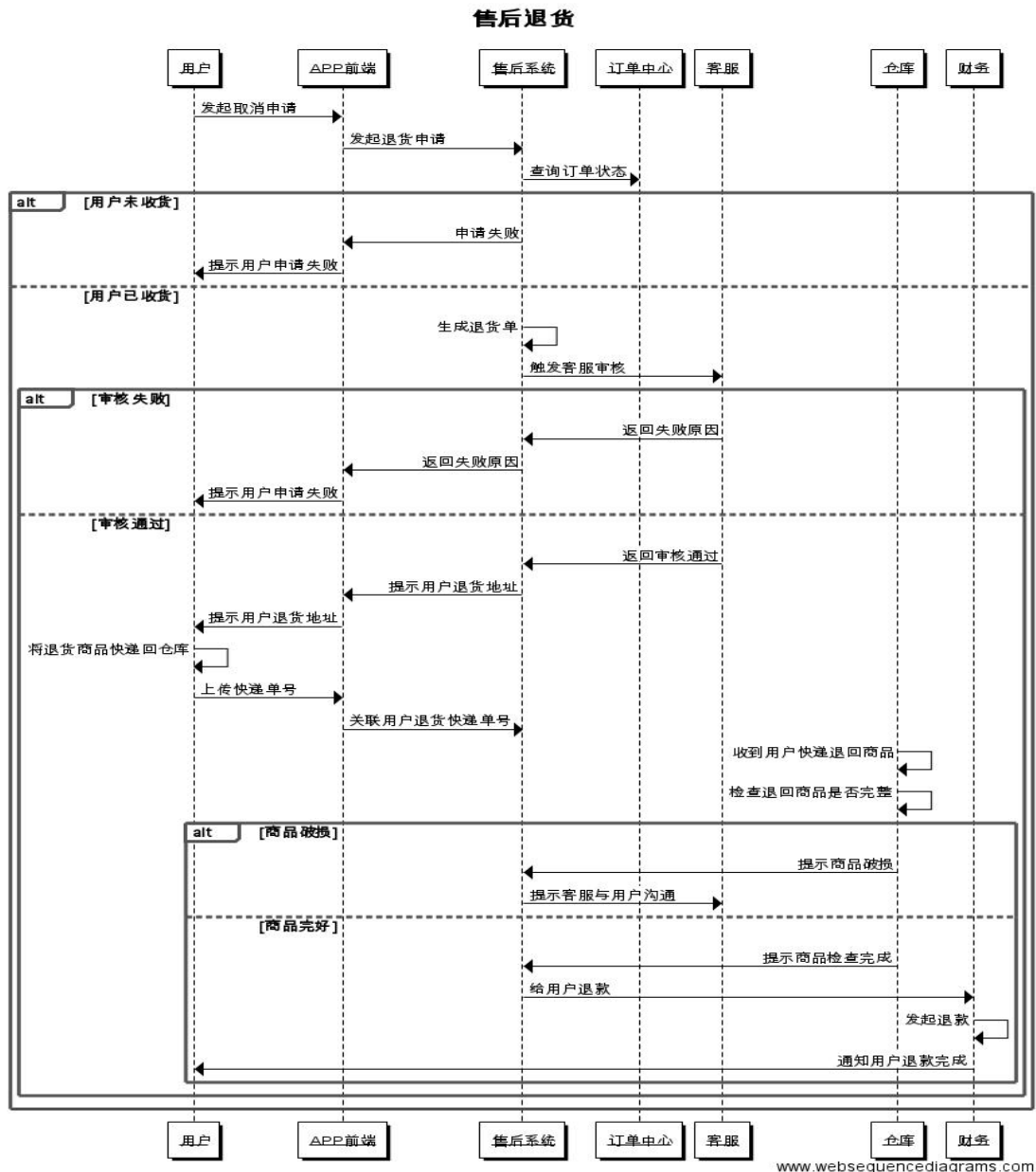
长时间未支付就取消订单, 实现取消订单的思路参考第八题

12、电商中超时支付怎么处理的

支付超时的也要重新生成订单, 重新发起支付。

13、退货怎么处理的

退货流程图



11、面对抢购 怎么处理并发

抢购系统也就是秒杀系统，要处理秒杀系统的并发问题，并不是某一个方面就能实现的，要多方面采取措施来实现。首先我们要清楚，高并发发生在哪些部分，然后正对不同的模块来进行优化。一般会发生在下图红色部分。



如何解决上面问题呢？主要从两个方面入手：

(1) 将请求尽量拦截在系统上游（不要让锁冲突落到数据库上去）。传统秒杀系统之所以挂，请求都压倒后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，流量虽大，下单成功的有效流量甚小。以 **12306** 为例，一趟火车其实只有 **2000** 张票，**200w** 个人来买，基本没有人能买成功，请求有效率为 **0**。

(2) 充分利用缓存，秒杀买票，这是一个典型的读多写少的应用场景，大部分请求是车次查询，票查询，下单和支付才是写请求。一趟火车其实只有 **2000** 张票，**200w** 个人来买，最多 **2000** 个人下单成功，其他人都是查询库存，写比例只有 **0.1%**，读比例占 **99.9%**，非常适合使用缓存来优化。好，后续讲讲怎么个“将请求尽量拦截在系统上游”法，以及怎么个“缓存”法，讲讲细节。

具体实施可以从几个方面考虑：

前端方面

1. 把详情页部署到 **CDN** 节点上

用户在秒杀开始之前，会对详情页做大量的刷新操作。所以我们将详情页部署到 **CDN** 上，**CDN** 将详情页做静态化处理。这样其实用户访问的静态页的 **html** 已经不在秒杀系统上了，而在 **CDN** 节点上。

2. 禁止重复提交请求、对用户请求限流

(a) 禁止重复提交

用户点击“查询”或者“购票”后，按钮置灰，禁止用户重复提交请求；

(b) 用户限流

限制用户在 x 秒之内只能提交一次请求；

后端方面

1. 根据用户 id 限制访问频率，对同样请求返回同一个页面

秒杀活动都会要求用户进行登录，对于同一个 **UID** 在 X 秒内发来的多次请求进行计数或者去重，限制在 X 秒内只有一个请求到达后端接口，其余的请求怎么办呢？使用**页面缓存**， X 秒内到达后端接口的请求，均返回同一页面。如此限流，既能保证用户有良好的用户体验（没有返回 **404**）又能保证系统的健壮性（利用页面缓存，把请求拦截在站点层了）。

2. 服务层进行优化

(a) **采用消息队列缓存请求**：既然服务层知道库存只有 **100** 台手机，那完全没有必要把 **100W** 个请求都传递到数据库，那么可以先把这些请求都写到消息队列缓存一下，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。

(b) **利用缓存应对读请求**：对类似于 **12306** 等购票业务，是典型的读多写少业务，大部分请求是查询请求，所以可以利用缓存(**Redis/Memcached**)分担数据库压力。

(c) **利用缓存应对写请求**：缓存也是可以应对写请求的，比如我们就可以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。

业务方面

1. 分时分段销售，将流量均摊

例如，原来统一 **10** 点，现在 **8** 点， **9** 点，...每隔半个小时放出一批

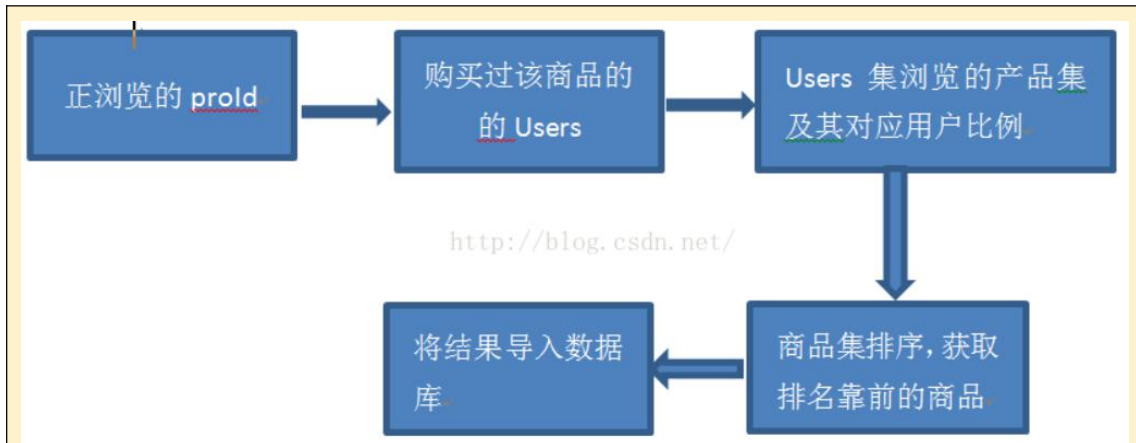
2. 请求多时，对数据粒度进行缓存

以 **12306** 购票为例子，你去购票，对于余票查询这个业务，票剩了 **58** 张，还是 **26** 张，你真的关注么，其实我们只关心有票和无票？流量大的时候，做一个粗粒度的“有票”“无票”缓存即可

3. 业务逻辑的异步处理

例如下单业务与支付业务的分离，下单成功后可以将支付业务单独异步处理，与抢购的业务分离，减少对业务处理的压力。

12、怎么根据 用户的浏览记录 进行推荐，说下推荐时 接口的思路是什么



主要思路如上图：

1. 根据用户正在浏览或添加到购物车的商品的 **id**,来过滤出购买过改商品的用 户
2. 根据购买过该商品的用户集合，在用户集合中根据每个用户的订单信息，查 找出该用户的订单商品信息，及用户的历史商品浏览信息
3. 根据所有用户的商品信息，找到所有的商品，然后对商品按照商品的销量进 行排序，把排名靠前的商品，写入缓存或者数据库
4. 当用户登录时，根据用户的 **id**，把上一步写入缓存或数据库的商品信息，取 出，推送给用户。

13、首页静态化是怎么做，为什么使用页面静态化而不是缓 存, 说下你的理由, 如果不是你选的 ,你会怎么做 说下理由; 局部刷新是怎么写的（定时任务）定时任务怎么实现 原理 是什么 为什么这样做,celery 怎么写的

静态化和缓存的主要区别在于

页面静态化是将数据库静态化到页面，客户端访问不需要查询数据库，主要存 放形式是静态化文件资源，存储于硬盘；

缓存是将数据存储在服务器内存，二者存放位置和形式不一样。

这二者使用主要看业务场景及网站优化的点，比如说秒杀的时候，肯会都要把也 页面进行静态化，放到 **CDN** 上面，这样能在前端就能抗住大量的并发请求；但 是在广告页面的广告数据我们就可以使用页面缓存来实现，同样不用对数据库进 行查询，只要访问内存就行。

定时任务的原理:

Cron 由 **crond** 守护进程和一组表 (**crontab** 文件) 组成。**crond** 守护进程是在系统启动时由 **init** 进程启动的, 受 **init** 进程的监视, 如果它不存在了, 会被 **init** 进程重新启动。这个守护进程每分钟唤醒一次, 并通过检查 **crontab** 文件(任务表) 判断需要做什么。

项目中使用定时任务的原因:

1. 在项目中我们的商品数据种类很多, 运营人员在修改商品数据的同时用户购买商品, 比如修改价格同时用户在购买, 会影响用户的体验;
2. 为了避免在修改商品价格时用户刚好下单购买造成的数据不一致问题, 我们把修改后的数据都用定时任务来执行, 并不会立马在前端展示, 而是在固定时间统一更新页面数据。

详情页的异步静态化处理--celery:

在管理后台中运营人员修改商品数据后, 点击保存或者修改按钮的同时触发异步任务的执行, 把修改后的数据通过异步任务处理为静态化数据。

14、业务需求是做音乐和视频的, 这样的前后端交互如何开发?

对于音乐、视频、图片这样上传和下载比较占用带宽的资源, 我们一般会存储到云端比如七牛云、阿里云, 或者会存放自己搭建的分布式文件存储系统中, 比如 **FastDFS** 上面, 来达到快速上传下载同时尽量不影响主要的业务服务器运行。

后端设计接口, 只要返回音乐或视频的 **url** 就可以了, 前端拿到 **url** 后根据 **url** 到指定服务器或者目录获取资源展示。具体实现与获取商品的图片思路一样。

15、支付模块的几种状态(待支付, 待发货等等)是平级的吗? 需不需要分级?

是平级的, 不用进行分级, 只是业务逻辑不一样的时候状态不一样而已

16、怎么监控日志文件，发邮件吗？云服务器已经封锁了相应端口，你怎么解决？

日志监控警告可以使用邮件也可以发短信。例如：监控订单信息，当累计订单量达到 **1000** 万时给全公司得人员发邮件，表示庆贺。

根据不同得操作系统再次打开就行了：如果是 **linux** 系统 而且是软件防火墙的话

应该是 **iptables** 命令 来配置端口；如果是 **wind** 参考 https://blog.csdn.net/spt_dream/article/details/75014619

17、使用 **elasticSearch** 搜索引擎为什么要建立分词

ES 是什么

Elasticsearch 是一个基于 **Lucene** 的实时的分布式搜索和分析引擎，它可以准实时地快速存储、搜索、分析海量的数据。

什么是全文检索

全文检索是指计算机索引程序通过扫描文章中的每一个词，对每一个词建立一个索引，指明该词在文章中出现的次数和位置，当用户查询时，检索程序就根据事先建立的索引进行查找，并将查找的结果反馈给用户的检索方式。这个过程类似于通过字典中的检索字表查字的过程。全文搜索搜索引擎数据库中的数据。

在全文搜索里，文档数据离不开搜索，而搜索离不开索引（没有索引搜索会很低效），倒排索引（**Inverted index**）是全文搜索系统里最高效的索引方法和数据结构，**ES** 的索引就是倒排索引。也称反向索引/置入索引或反向档案，用以存储一个映射：在全文搜索下某个单词在一个文档或者一组文档中的位置。

所以要在 **ElasticSearch** 中建立分词。

18、说一下你项目的数据库表结构

19、对于后台站点的用户活跃度统计 除了记录用户登陆时间进行筛选，还有没有别的方法

维护一张用户表，里面有 4 列：**guid, starttime, endtime, num**，分别是用户

的 **guid**，第一次访问时间，最后一次访问时间，访问天数；
表中 **guid** 是唯一的； 把当天数据去重后，与库中得历史记录进行 **join**，如果 **guid** 在历史库出现过，则将 **endtime** 更新为当天时间，**num** 加一； 否则，这是一个新用户，插入历史库，**starttime, endtime** 都为当天时间，**num** 初始值为 **1**。
只要 **num** 变化就说明用户是活跃得，