模拟冲刺记录

上线后QPS并发量，用户量、同时在线人数并发数等问题：

建议采用项目定制方案：项目给客户做的，甲方的数据我们拿不到。

但是要了解以下数据关键词：

（1）QPS（TPS）：每秒钟request/事务 数量

（2）并发数： 系统同时处理的request/事务数

（3）响应时间： 一般取平均响应时间

QPS（TPS）= 并发数/平均响应时间或者并发数 = QPS*平均响应时间 一个典型的上班签到系统，早上8点上班，7点半到8点的30分钟的时间里用户会登录签到系统进行签到。公司员工为1000人，平均每个员上登录签到系统的时长为5分钟。可以用下面的方法计算。 QPS =

1000/(30*60) 事务/秒 平均响应时间为 = 5*60 秒 并发数= QPS*平均响应时间 = 1000/(30*60) *(5*60)=166.7

说明：

一个系统吞吐量通常由QPS（TPS）、并发数两个因素决定，每套系统这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。

我们做项目要排计划，可以多人同时并发做多项任务，也可以一个人或者多个人串行工作，始终会有一条关键路径，这条路径就是项目的工期。系统一次调用的响应时间跟项目计划一样，也有一条关键路径，这个关键路径是就是系统影响时间； 关键路径是有CPU运算、IO、外部系统响应等等组成。

如果非要说，以下提供一套参数，但是仅供参考，可以根据自己设计适当调整：

用户总量几万+，日活 3000+，月活 12W+，一个月PV 30W+，并发量 500+

如何保证接口的幂等性?

1. 根据状态机很多时候业务表是有状态的，比如订单表中有：1–下单、2–已支付、3–完成、4–撤销等状态。如果这些状态的值是有规律的，按照业务节点正好是从小到大，我们就能通过它来保证接口的幂等性。假如id=123的订单状态是已支付，现在要变成完成状态。update `order` set status=3 where id=123 and status=2;第一次请求时，该订单的状态是已支付，值是2，所以该update语句可以正常更新数据，sql执行结果的影响行数是1，订单状态变成了3。后面有相同的请求过来，再执行相同的sql时，由于订单状态变成了3，再用status=2作为条件，无法查询出需要更新的数据，所以最终sql执行结果的影响行数是0，即不会真正的更新数据。但为了保证接口幂等性，影响行数是0时，接口也可以直接返回成功。
   具体步骤：
   1 用户通过浏览器发起请求，服务端收集数据。
   2 根据id和当前状态作为条件，更新成下一个状态
   3 判断操作影响行数，如果影响了1行，说明当前操作成功，可以进行其他数据操作。
   4 如果影响了0行，说明是重复请求，直接返回成功。
   主要特别注意的是，该方案仅限于要更新的表有状态字段，并且刚好要更新状态字段的这种特殊情况，并非所有场景都适用。
2. 加分布式锁其实前面介绍过的加唯一索引或者加防重表，本质是使用了数据库的分布式锁，也属于

分布式锁的一种。但由于数据库分布式锁的性能不太好，我们可以改用：redis或zookeeper。鉴于现在很多公司分布式配置中心改用apollo或nacos，已经很少用zookeeper了，我们以redis为例介绍分布式锁。目前主要有三种方式实现redis的分布式锁：

1 setNx命令

2 set命令

3 Redission框架

具体步骤：

1 用户通过浏览器发起请求，服务端会收集数据，并且生成订单号code作为唯一业务字段。

2 使用redis的set命令，将该订单code设置到redis中，同时设置超时时间。

3 判断是否设置成功，如果设置成功，说明是第一次请求，则进行数据操作。

4 如果设置失败，说明是重复请求，则直接返回成功。

3. 获取token

   除了上述方案之外，还有最后一种使用token的方案。该方案跟之前的所有方案都有点不一样，需要两次请求才能完成一次业务操作。

   第一次请求获取token

   第二次请求带着这个token，完成业务操作。

   具体步骤：

   1 用户访问页面时，浏览器自动发起获取token请求。

   2 服务端生成token，保存到redis中，然后返回给浏览器。

   3 用户通过浏览器发起请求时，携带该token。

   4 在redis中查询该token是否存在，如果不存在，说明是第一次请求，做则后续的数据操作。

   5 如果存在，说明是重复请求，则直接返回成功。

   6 在redis中token会在过期时间之后，被自动删除。

你们项目中有没有用到什么设计模式?

这个建议提前去了解几种常见的设计模式及其应用场景，将其套用到项目的某个场景中，以下提供几种常见的设计模式及其应用场景

1) 单例模式。

单例模式是一种常用的软件设计模式。

在它的核心结构中只包含一个被称为单例类的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。

应用场景：如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。

2) 工厂模式。

工厂模式主要是为创建对象提供了接口。

应用场景如下：

a、 在编码时不能预见需要创建哪种类的实例。

b、 系统不应依赖于产品类实例如何被创建、组合和表达的细节。

3) 策略模式。

策略模式：定义了算法族，分别封装起来，让它们之间可以互相替换。此模式让算法的变化独立于使用算法的客户。

应用场景如下。

a、 一件事情，有很多方案可以实现。

b、我可以在任何时候，决定采用哪一种实现。

c.、未来可能增加更多的方案。

d、 策略模式让方案的变化不会影响到使用方案的客户。

举例业务场景如下。

系统的操作都要有日志记录，通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面。日志写到数据库与文件中是两种算法，但调用方不关心，只负责写就是。

4) 观察者模式。

观察者模式又被称作发布/订阅模式，定义了对象间一对多依赖，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

应用场景如下：

a、对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。

b、对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节。

5) 迭代器模式。

迭代器模式提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。

应用场景如下：

当你需要访问一个聚集对象，而且不管这些对象是什么都需要遍历的时候，就应该考虑用迭代器模式。其实stl容器就是很好的迭代器模式的例子。

6) 模板方法模式。

模板方法模式定义一个操作中的算法的骨架，将一些步骤延迟到子类中，模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些步骤。

应用场景如下：

对于一些功能，在不同的对象身上展示不同的作用，但是功能的框架是一样的。

ThreadLocal 的原理（高薪常问）

ThreadLocal：为共享变量在每个线程中创建一个副本，每个线程都可以访问自己内部的副本变量。通过threadlocal 保证线程的安全性。

其实在ThreadLocal 类中有一个静态内部类 ThreadLocalMap(其类似于 Map)，用键值对的形式存储每一个线程的变量副本，ThreadLocalMap 中元素的 key 为当前 ThreadLocal 对象，而 value 对应线程的变量副本。

ThreadLocal 本身并不存储值，它只是作为一个 key 保存到 ThreadLocalMap 中，但是这里要注意的是它作为一个key 用的是弱引用，因为没有强引用链，弱引用在 GC 的时候可能会被回收。这样就会在ThreadLocalMap 中存在一些 key 为 null 的键值对（Entry）。因为 key 变成 null 了，我们是没法访问这些 Entry 的，但是这些 Entry 本身是不会被清除的。如果没有手动删除对应key 就会导致这块内存即不会回收也无法访问，也就是内存泄漏。

使用完ThreadLocal 之后，记得调用 remove 方法。 在不使用线程池的前提下，即使不调用remove 方法，线程的"变量副本"也会被 gc 回收，即不会造成内存泄漏的情况。

同步锁、死锁、乐观锁、悲观锁（高薪常问）

同步锁：

当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。Java 中可以使用 synchronized 关键字来取得一个对象的同步锁。

死锁：

何为死锁，就是多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。

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_conditio 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。

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。

反射

在Java 中的反射机制是指在运行状态中，对于任意一个类都能够知道这个类所有的
属性和方法；并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息
以及动态调用对象方法的功能成为Java 语言的反射机制。

获取Class 对象的 3 种方法 ：

调用某个对象的getClass()方法

Person p=new Person();

Class clazz=p.getClass();

调用某个类的class 属性来获取该类对应的 Class 对象

Class clazz=Person.class;

使用Class 类中的 forName()静态方法(最安全/性能最好)

Class clazz=Class.forName("类的全路径"); (最常用)

Hashmap 和 hashtable ConcurrentHashMap 区别（高薪常问）

区别对比一(HashMap 和 HashTable 区别)：

1、HashMap 是非线程安全的，HashTable 是线程安全的。

2、HashMap 的键和值都允许有 null 值存在，而 HashTable 则不行。

3、因为线程安全的问题，HashMap 效率比 HashTable 的要高。

4、Hashtable 是同步的，而 HashMap 不是。因此，HashMap 更适合于单线程环境，而Hashtable 适合于多线程环境。一般现在不建议用 HashTable, ①是HashTable 是遗留类，内部实现很多没优化和冗余。②即使在多线程环境下，现在也有同步的ConcurrentHashMap 替代，没有必要因为是多线程而用HashTable。

区别对比二(HashTable 和 ConcurrentHashMap 区别)：

HashTable 使用的是 Synchronized 关键字修饰，ConcurrentHashMap 是JDK1.7 使用了锁分段技术来保证线程安全的。JDK1.8ConcurrentHashMap 取消了Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8的结构类似，数组+链表/红黑二叉树。

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升N 倍。

Hashmap 的底层原理
HashMap 在 JDK1.8 之前的实现方式 数组+链表，
但是在 JDK1.8 后对 HashMap 进行了底层优化,改为了由 数组+链表或者数值+红黑树实现,主要的目的是提高查找效率

1. Jdk8 数组+链表或者数组+红黑树实现，当链表中的元素超过了 8 个以后， 会将链表转换为红黑树，当红黑树节点 小于 等于 6 时又会退化为链表。
2. 当 new HashMap():底层没有创建数组，首次调用 put()方法示时，底层创建长度为 16 的数组，
   jdk8 底层的数组是：Node[],而非 Entry[]，用数组容量大小乘以加载因子得到一个值，一旦数组中存储的元素个数超过该值就会调用 rehash 方法将数组容量增加到原来的两倍，专业术语叫做扩容，在做扩容的时候会生成一个新的数组，原来的所有数据需要重新计算哈希码值重新分配到新的数组，所以扩容的操作非常消耗性能.
   默认的负载因子大小为 0.75，数组大小为 16。也就是说，默认情况下，那么当 HashMap 中元素个数超过 16*0.75=12 的时候，就把数组的大小扩展为 2*16=32，即扩大一倍。
3. 在我们 Java 中任何对象都有 hashcode，hash 算法就是通过 hashcode 与自己进行向右位移 16 的异或运算。这样做是为了计算出来的 hash 值足够随机，足够分散，还有产生的数组下标足够随机，
   map.put(k,v)实现原理
   （
   1）首先将 k,v 封装到 Node 对象当中（节点）。
   （
   2）先调用 k 的 hashCode()方法得出哈希值，并通过哈希算法转换成数组的下标。
   （
   3）下标位置上如果没有任何元素，就把 Node 添加到这个位置上。如果说下标对应的位置上有链表。此时，就会拿着 k 和链表上每个节点的 k 进行 equal。如果所有的 equals 方法返回都是 false，那么这个新的节点将被添加到链表的末尾。如其中有一个 equals 返回了 true，那么这个节点的 value 将会被覆盖。
   map.get(k)实现原理
   (1)、先调用 k 的 hashCode()方法得出哈希值，并通过哈希算法转换成数组的下标。

(2)、在通过数组下标快速定位到某个位置上。重点理解如果这个位置上什么都没有，则返回 null。如果这个位置上有单向链表，那么它就会拿着参数 K 和单向链表上的每一个节点的 K 进行 equals，如果所有 equals 方法都返回 false，则 get 方法返回 null。如果其中一个节点的 K 和参数 K 进行 equals 返回 true，那么此时该节点的 value 就是我们要找的 value 了，get 方法最终返回这个要找的 value。

4. Hash 冲突
   不同的对象算出来的数组下标是相同的这样就会产生 hash 冲突，当单线链表达到一定长度后效率会非常低。
5. 在链表长度大于 8 的时候，将链表就会变成红黑树，提高查询的效率。

存储引擎

1.MyISAM 存储引擎

主要特点：

MySQL5.5 版本之前的默认存储引擎

支持表级锁（表级锁是MySQL 中锁定粒度最大的一种锁，表示对当前操作的整张表加锁）；

不支持事务，外键。

适用场景：对事务的完整性没有要求，或以select、insert 为主的应用基本都可以选用

MYISAM。在 Web、数据仓库中应用广泛。

特点：

1、不支持事务、外键

2、每个 myisam 在磁盘上存储为 3 个文件，文件名和表名相同，扩展名分别是

.frm

--------存储表定义

.MYD ---------MYData，存储数据

.MYI

---------MYIndex，存储索引

2.InnoDB 存储引擎

主要特点：

MySQL5.5 版本之后的默认存储引擎；

支持事务；

支持行级锁（行级锁是Mysql 中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁）；

支持聚集索引方式存储数据。

如何避免索引失效

(1) 范围查询, 右边的列不能使用索引, 否则右边的索引也会失效.

索引生效案例

```
select * from tb_seller where name = "小米科技" and status = "1" and address = "北京市";
```

```
select * from tb_seller where name = "小米科技" and status >= "1" and address = "北京市";
```

索引失效案例

```
select * from tb_seller where name = "小米科技" and status > "1" and address = "北京市";
```

address 索引失效, 因为 status 是大于号, 范围查询.

(2) 不要在索引上使用运算, 否则索引也会失效.

比如在索引上使用切割函数, 就会使索引失效.

```
select * from tb_seller where substring(name, 3, 2) = "科技";
```

39**(3) 字符串不加引号, 造成索引失效.**

如果索引列是字符串类型的整数, 条件查询的时候不加引号会造成索引失效. Mysql 内置的优化会有隐式转换.

索引失效案例

```
select * from tb_seller where name = "小米科技" and status = 1
```

(4) 尽量使用覆盖索引, 避免 select *, 这样能提高查询效率.

如果索引列完全包含查询列, 那么查询的时候把要查的列写出来, 不使用 select *

```
select sellerid, name, status from tb_seller where name = "小米科技" and staus = "1"
and address = "西安市";
```

(5) or 关键字连接

用or 分割开的条件, 如果 or 前面的列有索引, or 后面的列没有索引, 那么查询的时候前后索引都会失效

如果一定要用or 查询, 可以考虑下 or 连接的条件列都加索引, 这样就不会失效了.

Sql 语句调优

根据业务场景建立复合索引只查询业务需要的字段，如果这些字段被索引覆盖，将极大的提高查询效率.

|

多表连接的字段上需要建立索引，这样可以极大提高表连接的效率.

|

where 条件字段上需要建立索引, 但 Where 条件上不要使用运算函数，以免索引失效.

|

排序字段上, 因为排序效率低, 添加索引能提高查询效率.

|

优化insert 语句: 批量列插入数据要比单个列插入数据效率高.

|

优化order by 语句: 在使用 order by 语句时, 不要使用 select *, select 后面要查有索引的列, 如果一条 sql 语句中对多个列进行排序, 在业务允许情况下, 尽量同时用升序或同时用降序.

l

优化group by 语句: 在我们对某一个字段进行分组的时候, Mysql 默认就进行了排序,

但是排序并不是我们业务所需的, 额外的排序会降低效率. 所以在用的时候可以禁止

排序, 使用 order by null 禁用.

select age, count(*) from emp group by age order by null

l 尽量避免子查询, 可以将子查询优化为 join 多表连接查询

微服务网关中如何实现限流?

令牌桶算法是比较常见的限流算法之一，大概描述如下：

1）所有的请求在处理之前都需要拿到一个可用的令牌才会被处理；

2）根据限流大小，设置按照一定的速率往桶里添加令牌；

3）桶设置最大的放置令牌限制，当桶满时、新添加的令牌就被丢弃或者拒绝；

4）请求达到后首先要获取令牌桶中的令牌，拿着令牌才可以进行其他的业务逻辑，处理完业务逻辑之后，将令牌直接删

除；

5）令牌桶有最低限额，当桶中的令牌达到最低限额的时候，请求处理完之后将不会删除令牌，以此保证足够的限流