```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
       //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

```
public class Singleton {

    //声明为 private 避免调用默认构造方法创建对象
    private Singleton() {
    }

    //声明为 private 表明静态内部该类只能在该 Singleton 类中被访问
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getUniqueInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int CACHE_SIZE;

    /**
     * 传递进来最多能缓存多少数据
     *
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {
        // true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的放在尾部。
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        CACHE_SIZE = cacheSize;
    }

```

```
    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // 当 map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。
        return size() > CACHE_SIZE;
    }
}
```

基础:

集合:

        - 死亡（Terminated）
    - 2. Java实现多线程的方式及三种方式的区别
        - 实现 Runnable 接口
        - 实现 Callable 接口
        - 继承 Thread 类
        - 实现接口 VS 继承 Thread
        - 三种方式的区别
    - 3. 基础线程机制
        - Executor
        - Daemon（守护线程）
        - sleep()
        - yield()
        - 线程阻塞
    - 《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

```
Executors 返回线程池对象的弊端如下：
FixedThreadPool 和 SingleThreadExecutor
：  允许请求的队列长度为 Integer.MAX_VALUE ，可能堆积大量的请求，从而导致OOM。
CachedThreadPool 和 ScheduledThreadPool
：  允许创建的线程数量为 Integer.MAX_VALUE ，可能会创建大量线程，从而导致OOM。
```

        - 4. 中断
        - InterruptedException
        - interrupted()
        - Executor 的中断操作
    - 5. 互斥同步
        - synchronized
        - ReentrantLock 。 补：原理
        - synchronized 和 ReentrantLock 比较
        - synchronized与lock的区别，使用场景。看过synchronized的源码没?
        - 并发编程面试必备：synchronized 关键字使用、底层原理、JDK1.6 之后的底层优化以及 和 ReenTrantLock 的对比
        - 什么是CAS
            - 入门例子
            - Compare And Swap
        - 什么是乐观锁和悲观锁
        - Synchronized（对象锁）和Static Synchronized（类锁）区别
    - 6. 线程之间的协作
        - join()
        - wait() notify() notifyAll()
        - await() signal() signalAll()
        - sleep和wait有什么区别
    - 7. J.U.C - AQS 。 并发编程面试必备：AQS 原理以及 AQS 同步组件总结
        - CountdownLatch
        - CyclicBarrier
        - Semaphore
        - 总结
    - 8. J.U.C - 其它组件

        - 锁的公平性
        - 在 finally 语句中调用 unlock()
    - 12. 并发包(J.U.C)下面，都用过什么
    - 13. 从volatile说到,i++原子操作,线程安全问题
- 参考资料
- 更新日志

补充：

线程池-ThreadPoolExecute源码分析

java中的notify和notifyAll有什么区别?

Java IO

- 1、磁盘操作（File）
- 2、字节操作（*Stream）
- 3、字符操作（*Reader | *Writer）
- 4、Java序列化，如何实现序列化和反序列化，常见的序列化协议有哪些?
    - Java序列化定义
    - 如何实现序列化和反序列化，底层怎么实现
    - 相关注意事项
    - 常见的序列化协议有哪些
- 5、同步和异步
- 6、Java中的NIO，BIO，AIO分别是什么
    - BIO
    - NIO
    - AIO (NIO.2)
    - 总结
- 7、BIO，NIO，AIO区别 。 linux的io模型/aio
- 8、Stock通信的伪代码实现流程
- 9、网络操作
    - InetAddress
    - URL
    - Sockets
    - Datagram
    - 什么是Socket?

JVM：

- 前言
- 核心知识
    - JVM体系结构
        - JVM各个模块简介
        - JVM是如何工作的?
    - 1. 运行时数据区域
        - 1. 程序计数器（线程私有）
        - 2. 虚拟机栈（线程私有）
        - 3. 本地方法栈（线程私有）
        - 4. 堆
            - 新生代 （Young Generation）

补充:

JAVA WEB:

补充：

- get和post请求的区别
- 什么情况下调用doGet()和doPost()
- spring中的设计模式
- Bean 的作用域
- Spring中bean的作用域与生命周期
- bean的加载过程

MYSQL:

1NF：每一列都是不可分割的原子数据项

2NF: 要求实体的属性完全依赖于主关键字。

3NF: 在2NF基础上，任何非主属性不依赖于其它非主属性

巴斯-科德范式（BCNF）： 在3NF基础上，任何非主属性不能对主键子集依赖

数据库设计的三大范式

补充：

- 一千行MySQL学习笔记
- **深入学习MySQL事务：ACID特性的实现原理**
- 聚合函数 注意：where子句中不能使用聚合函数，因为聚合函数对结果集进行操作，而where子句运行时还没有筛选出结果集，
- 所以此时使用聚合函数会报错；与where相比，having虽然也是用来筛选的，但having是用来筛选分组的，
- 跟在group by之后，所以having子句运行时结果集已经被筛选出，此时可以使用聚合函数进行二次筛选。
- MySql面试前必须练习到熟练的
- 添加索引
- 创建索引
- sql优化方案

操作系统：

        - 1. which
        - 2. whereis
        - 3. locate
        - 4. find
        - *. grep的使用，一定要掌握，每次都会问在文件中查找（包含匹配）
        - *. 管道
    - 11. 压缩与解压缩命令
        - .zip
        - .gz
        - .bz2
        - tar
        - .tar.gz
        - .tar.bz2
    - 12. Bash
        - 特性
        - 变量操作
        - 指令搜索顺序
        - 输出重定向
        - 输入重定向
    - 13. 正则表达式
        - cut
        - grep
        - printf
        - awk
        - sed
    - 14. 进程管理
        - 查看进程
            - 1. ps
            - 2. top
            - 3. pstree
            - 4. netstat
        - 进程状态
            - SIGCHLD
            - wait()
            - waitpid()
            - 孤儿进程
            - 僵尸进程
    - 15. 进程和线程的区别
    - 16. kill用法，某个进程杀不掉的原因（进入内核态，忽略kill信号）
    - 17. 包管理工具
        - 软件类型
        - 发行版
    - 18. 网络配置和网络诊断命令
    - 19. 磁盘管理 df
    - **Linux中 du （详解）和 df （详解）以及它们的区别**
    - 20. VIM 三个模式
    - 21. 用户管理

补充：

五种io模型

计算机网络：

- 11. Cookie 与 Session 选择

  session 和 cookie的区别

  session cookie都是会话跟踪技术。cookie通过在客户端记录信息，确认用户身份，session通过在服务器端记录信息确定用户身份。但是session的实现依赖于Cookie ， sessionId（session的唯一标识需要存放在服务器上）。

  区别

  1.cookie数据数据存放在客户的浏览器上，session数据存放在服务器上

  2.cookie不是很安全，别人可以分析存放在本地的cookie并进行cookie欺骗，考虑到安全应当使用session

  3.session会在一定时间内保存在服务器上，当访问增多时，会比较占用服务器性能，考虑到减轻服务器性能方面，应当使用cookie。

  4.单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

  建议：将登陆信息重要信息放在session中 其它信息保存在cookie中

  - 8. tomcat和Servlet的联系
- 12. HTTPs安全性
  - （1）对称密钥加密
  - （2）非对称密钥加密
  - （3）HTTPs 采用的加密方式

    1. 首先，客户端 A 访问服务器 B ，比如我们用浏览器打开一个网页 www.baidu.com ，这时，浏览器就是客户端 A ，百度的服务器就是服务器 B 了。这时候客户端 A 会生成一个随机数1，把随机数1 、自己支持的 SSL 版本号以及加密算法等这些信息告诉服务器 B 。
    2. 服务器 B 知道这些信息后，然后确认一下双方的加密算法，然后服务端也生成一个随机数 B ，并将随机数 B 和 CA 颁发给自己的证书一同返回给客户端 A 。
    3. 客户端 A 得到 CA 证书后，会去校验该 CA 证书的有效性，校验方法在上面已经说过了。校验通过后，客户端生成一个随机数3 ，然后用证书中的公钥加密随机数3 并传输给服务端 B 。
    4. 服务端 B 得到加密后的随机数3，然后利用私钥进行解密，得到真正的随机数3。
    5. 最后，客户端 A 和服务端 B 都有随机数1、随机数2、随机数3，然后双方利用这三个随机数生成一个对话密钥。之后传输内容就是利用对话密钥来进行加解密了。这时就是利用了对称加密，一般用的都是 AES 算法。
    6. 客户端 A 通知服务端 B ，指明后面的通讯用对话密钥来完成，同时通知服务器 B 客户端 A 的握手过程结束。
    7. 服务端 B 通知客户端 A，指明后面的通讯用对话密钥来完成，同时通知客户端 A 服务器 B 的握手过程结束。
    8. SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户端 A 和服务器 B 开始使用相同的对话密钥进行数据通讯。

    到此，SSL 握手过程就讲完了。可能上面的流程太过于复杂，我们简单地来讲：

    1. 客户端和服务端建立 SSL 握手，客户端通过 CA 证书来确认服务端的身份；
    2. 互相传递三个随机数，之后通过这随机数来生成一个密钥；
    3. 互相确认密钥，然后握手结束；
    4. 数据通讯开始，都使用同一个对话密钥来加解密；

- 13. SSL/TLS协议的握手过程
  - SSL（Secure Socket Layer，安全套接字层）
  - TLS（Transport Layer Security，传输层安全协议）
  - （1）client hello
  - （2）server hello
  - （3）server certificate
  - （4）Server Hello Done
  - （5）Client Key Exchange
  - （6）Change Cipher Spec(Client)
  - （7）Finished(Client)

        - （8）Change Cipher Spec(Server)
        - （9）Finished(Server)
        - （10-11）Application Data
        - （12）Alert：warning, close notify
        - （*）demand client certificate
        - （*）check server certificate
    - 14. 数字签名、数字证书、SSL、https是什么关系?
        - 密码
        - 密钥
        - 对称加密
        - 公钥加密（非对称加密）
        - 消息摘要
        - 消息认证码
        - 数字签名
        - 公钥证书
    - 15. HTTP和HTTPS的区别【阿里面经OneNote】
    - 16. HTTP2.0特性
        - （1）二进制分帧
        - （2）多路复用
        - （3）服务器推送
        - （4）头部压缩
- 第三部分：网络层
    - 1. mac和ip怎么转换
    - 2. IP地址子网划分
    - 3. 地址解析协议ARP
    - 4. 交换机和路由器的区别
    - 5. 子网掩码的作用
- 附录：参考资料

spring 源码

- spring-core
- spring-aop
- spring-context
- spring-task
- spring-transaction
- spring-mvc
- guava-cache

epoll的ET和LT模式

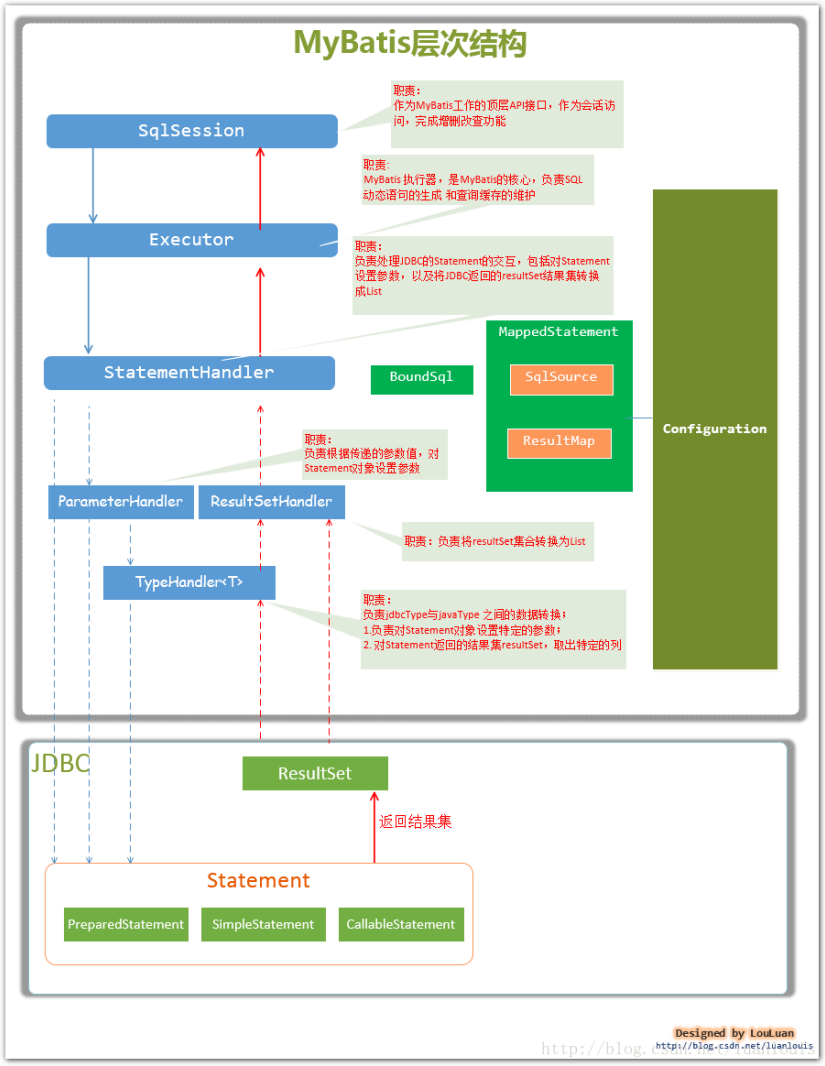
epoll 对文件的描述符的操作有两种模式： LT(Level Trigger, 电平触发)模式 和 ET(Edge Trigger ,边沿触发)模式。LT模式是默认的工作模式，这个模式下epoll相当于一个效率较高的poll。当往epoll中内核事件表中注册EPOLLET事件时，epoll将以ET模式来操作该文件描述符。ET是epoll的高效模式。

对于采用LT工作的文件描述符，当epolll_wait检测到其上有事件发生并将此事件通知应用程序后，应用程序可以不立即处理该事件。这样当应用程序下次调用epoll_wait时，epolll_wait还会再次向应用程序通知此事件，直到有该事件被处理。而对于采用ET模式的文件描述符，当epoll_wait检测当其上有事件发生时并将此事件通知应用程序后，应用程序必须立即处理该事件，因为后序的epolll_wait调用不再讲此事件通知应用程序，可见，ET模式在很大程度上降底了同一个epoll事件被重复触发的次数，因此效率要比LT模式高。

epoll的ET怎么保证读完所有的数据
什么是拥塞控制，什么是流量控制?

操作系统层面的线程通信共享内存，信号量，应该跟进程通信差不多吧，用udp实现tcp，这个学计网的时候就做过实验，无非就是每个消息都加上一个系列号，接收端保持着上一个已经接收的系列号等着下面的系列号，发送端发送一个消息带上序列号并且等待ack，设置超时重传，有人说三次握手四次握手这个应该不相关吧，毕竟重点在可靠数据传输而不是建立连接，这方面可以看看tcp的滑动窗口协议。QQ不发送重复消息，这个应该也是跟ack跟序列号相关的，我发给你一个消息，期待收到ack，没收到就重传，你发送的ack如果丢失了收到重复消息了，那就丢掉消息，再发ack，怎么确定是重复消息呢，还是系列号来定

Mybatis
MyBatis框架及原理分析



[《深入理解mybatis原理》 MyBatis的架构设计以及实例分析]
Mybatis的整体架构分为三层，分别是基础支持层、核心处理层和接口层

个人认为的关键几个模块：
基础支持层：
- 类型转换模块：mybatis为简化配置文件提供别名机制，该机制是类型转换的主要功能之一，在为sql语句绑定实参时，会将数据由java类型转换成jdbc类型，在映射结果集的时候，会将数据由jdbc类型转换为java类型。
- 解释器模块：主要提供几个功能，一个是对xpath进行封装，为mybatis-config.xml配置文件以及映射配置文件提供支持；另一功能是为处理动态sql语句中的占位符提供支持。
- 缓存模块: 一级缓存和二级缓存，一级缓存是基于sqlsession的二级缓存是基于mapper的。mybatis中自带的这两级缓存与mybatis以及整个应用都是运行在同一个jvm当中，共享一块堆内存。如果这两级缓存中的数据量比较大，则可能影响系统中其它功能的使用。所以当需要缓存大量数据时，优先考虑使用redis Memcache。

核心处理层：

* 配置解析：在mybatis初始化过程中，会加载mybatis-config.xml配置文件，映射配置文件以及mapper接口中的注解信息，解析后的配置信息会形成对应的对象并保存到Configuration对象中，利用该Configuration对象创建SqlsessionFactory对象。待Mybatis初始化后，可以通过初始化得到SqlSessionFactory创建Sqlsession对象并完成数据库操作。
解析mybatis-config.xml -> 形成对象 -> 保存到configuration对象中 -> 用configuration对象创建SqlSessionFactory -> 创建sqlsession

* sql执行：sql执行涉及多个组件，比较重要的是Executor、StatementHandler 、ParameterHandler、ResultSetHandler。Executor主要负责维护一级缓存和二级缓存，并提供事务管理的相关操作，它会将数据库相关操作委托给StatementHandler完成。
StatementHandler首先通过ParameterHandler完成sql语言的实参绑定，然后通过java.sql.Statement对象执行sql语句得到结果集，最后通过ResultSetHandler完成结果集的映射，得到结果对象并返回。
美团技术: 聊聊MyBatis缓存机制

一级缓存总结：
- 1.MyBatis一级缓存的生命周期和SqlSession一致。
- 2.MyBatis一级缓存内部设计简单，只是一个没有容量限定的HashMap，在缓存的功能性上有所欠缺。
- 3.MyBatis的一级缓存最大范围是SqlSession内部，有多个SqlSession或者分布式的环境下，数据库写操作会引起脏数据，建议设定缓存级别为Statement。
二级缓存总结：
- 1.MyBatis的二级缓存相对于一级缓存来说，实现了SqlSession之间缓存数据的共享，同时粒度更加的细，能够到namespace级别，通过Cache接口实现类不同的组合，对Cache的可控性也更强。
- 2.MyBatis在多表查询时，极大可能会出现脏数据，有设计上的缺陷，安全使用二级缓存的条件比较苛刻。
- 3.在分布式环境下，由于默认的MyBatis Cache实现都是基于本地的，分布式环境下必然会出现读取到脏数据，需要使用集中式缓存

将MyBatis的Cache接口实现，有一定的开发成本，直接使用Redis、Memcached等分布式缓存可能成本更低，安全性也更高。

1 视图是一种虚拟的表，是从数据库中一个或者多个表中导出来的表。 2，数据库中只存放了视图的定义，而并没有存放视图中的数据，这些数据存放在原来的表中。 3，使用视图查询数据时，数据库系统会从原来的表中取出对应的数据

1，使操作简便化; 2，增加数据的安全性; 3，提高表的逻辑独立性;

springcloud:

springcloud 原理： https://juejin.im/post/5be13b83f265da6116393fc7

SpringCloud源码基本原理学习

源码篇：

- 深入理解Eureka之源码解析
- 深入理解Ribbon之源码解析
- 深入理解Feign之源码解析
- 深入理解Hystrix之文档翻译
- 深入理解Zuul之源码解析

拓展知识：

高并发架构

消息队列

- 为什么使用消息队列？消息队列有什么优点和缺点？Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么优点和缺点？
- 如何保证消息队列的高可用？
- 如何保证消息不被重复消费？（如何保证消息消费的幂等性）
- 如何保证消息的可靠性传输？（如何处理消息丢失的问题）
- 如何保证消息的顺序性？
- 如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？
- 如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路。

搜索引擎

- es 的分布式架构原理能说一下么（es 是如何实现分布式的啊）？
- es 写入数据的工作原理是什么啊？es 查询数据的工作原理是什么啊？底层的 lucene 介绍一下呗？倒排索引了解吗？
- es 在数据量很大的情况下（数十亿级别）如何提高查询效率啊？
- es 生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？

缓存

- 在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？
- Redis 和 Memcached 有什么区别？Redis 的线程模型是什么？为什么单线程的 Redis 比多线程的 Memcached 效率要高得多？
- Redis 都有哪些数据类型？分别在哪些场景下使用比较合适？
- Redis 的过期策略都有哪些？手写一下 LRU 代码实现？
- 如何保证 Redis 高并发、高可用？Redis 的主从复制原理能介绍一下么？Redis 的哨兵原理能介绍一下么？
- Redis 的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？

- Redis 集群模式的工作原理能说一下么？在集群模式下，Redis 的 key 是如何寻址的？分布式寻址都有哪些算法？了解一致性 hash 算法吗？如何动态增加和删除一个节点？
- 了解什么是 redis 的雪崩、穿透和击穿？Redis 崩溃之后会怎么样？系统该如何应对这种情况？如何处理 Redis 的穿透？
- 如何保证缓存与数据库的双写一致性？
- Redis 的并发竞争问题是什么？如何解决这个问题？了解 Redis 事务的 CAS 方案吗？
- 生产环境中的 Redis 是怎么部署的？

分库分表

- 为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？你们具体是如何对数据库如何进行垂直拆分或水平拆分的？
- 现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？
- 如何设计可以动态扩容缩容的分库分表方案？
- 分库分表之后，id 主键如何处理？

读写分离

- 如何实现 MySQL 的读写分离？MySQL 主从复制原理是啥？如何解决 MySQL 主从同步的延时问题？

高并发系统

- 如何设计一个高并发系统？

分布式系统

面试连环炮

系统拆分

- 为什么要进行系统拆分？如何进行系统拆分？拆分后不用 Dubbo 可以吗？

分布式服务框架

- 说一下 Dubbo 的工作原理？注册中心挂了可以继续通信吗？
- Dubbo 支持哪些序列化协议？说一下 Hessian 的数据结构？PB 知道吗？为什么 PB 的效率是最高的？
- Dubbo 负载均衡策略和集群容错策略都有哪些？动态代理策略呢？
- Dubbo 的 spi 思想是什么？
- 如何基于 Dubbo 进行服务治理、服务降级、失败重试以及超时重试？
- 分布式服务接口的幂等性如何设计（比如不能重复扣款）？
- 分布式服务接口请求的顺序性如何保证？
- 如何自己设计一个类似 Dubbo 的 RPC 框架？

分布式锁

- Zookeeper 都有哪些应用场景？
- 使用 Redis 如何设计分布式锁？使用 Zookeeper 来设计分布式锁可以吗？以上两种分布式锁的实现方式哪种效率比较高？

分布式事务

- 分布式事务了解吗？你们如何解决分布式事务问题的？TCC 如果出现网络连不通怎么办？XA 的一致性如何保证？

分布式会话

- 集群部署时的分布式 Session 如何实现？

高可用架构

- Hystrix 介绍
- 电商网站详情页系统架构
- Hystrix 线程池技术实现资源隔离
- Hystrix 信号量机制实现资源隔离
- Hystrix 隔离策略细粒度控制

- 深入 Hystrix 执行时内部原理
- 基于 request cache 请求缓存技术优化批量商品数据查询接口
- 基于本地缓存的 fallback 降级机制
- 深入 Hystrix 断路器执行原理
- 深入 Hystrix 线程池隔离与接口限流
- 基于 timeout 机制为服务接口调用超时提供安全保护

秒杀系统

架构设计原则

1.数据尽可能少

1.网络传输需要时间

2.数据需要服务器处理，而服务器通常都要做压缩和字符编码

3.要求系统依赖的数据能少就少，包括系统完成某些业务逻辑需要读取和保存的数据，调用其他服务会涉及数据的序列化和反序列化

2. 请求数要尽量少

额外请求应该尽量少，比如说，这个页面依赖的 CSS/JavaScript、图片，以及 Ajax 请求等等都定义为“额外请求”。减少请求数最常用的一个实践就是合并 CSS 和 JavaScript 文件，把多个 JavaScript 文件合并成一个文件，在 URL 中用逗号隔开

3. 路径要尽量短：就是用户发出请求到返回数据这个过程中，需求经过的中间的节点数。

通常，这些节点可以表示为一个系统或者一个新的 Socket 连接（比如代理服务器只是创建一个新的 Socket 连接来转发请求）。每经过一个节点，一般都会产生一个新的 Socket 连接。

缩短请求路径不仅可以增加可用性，同样可以有效提升性能（减少中间节点可以减少数据的序列化与反序列化），并减少延时（可以减少网络传输耗时）

要缩短访问路径有一种办法，就是多个相互强依赖的应用合并部署在一起，把远程过程调用（RPC）变成 JVM 内部之间的方法调用。

4. 依赖要尽量少：指的是要完成一次用户请求必须依赖的系统或者服务，这里的依赖指的是强依赖。

要减少依赖，我们可以给系统进行分级，比如 0 级系统、1 级系统、2 级系统、3 级系统，0 级系统如果是最重要的系统，那么 0 级系统强依赖的系统也同样是最重要的系统，以此类推。

注意，0 级系统要尽量减少对 1 级系统的强依赖，防止重要的系统被不重要的系统拖垮。

5. 不要有单点

单点意味着没有备份，风险不可控，设计分布式系统最重要的原则就是“消除单点”。

避免将服务的状态和机器绑定，即把服务无状态化，这样服务就可以在机器中随意移动。

例如把和机器相关的配置动态化，这些参数可以通过配置中心来动态推送，在服务启动时动态拉取下来，我们在这些配置中心设置一些规则来方便地改变这些映射关系。springcloud 的config

动静分离

“动态数据”和“静态数据”的主要区别就是看页面中输出的数据是否和 URL、浏览者、时间、地域相关，以及是否含有 Cookie 等私密数据。

静态数据做缓存

第一，你应该把静态数据缓存到离用户最近的地方。用户浏览器里、CDN 上或者在服务端的 Cache 中。

第二，静态化改造就是要直接缓存 HTTP 连接。

第三，让在 Web 服务器层上做缓存静态数据也很重要。

Java 系统本身也有其弱点（比如不擅长处理大量连接请求，每个连接消耗的内存较多，Servlet 容器解析 HTTP 协议较慢）

动态内容

1.URL 唯一化。

2.分离浏览者相关的因素。浏览者相关的因素包括是否已登录，以及登录身份等，这些相关因素我们可以单独拆分出来，通过动态请求来获取。

3.分离时间因素。服务端输出的时间也通过动态请求获取。

4.异步化地域因素。详情页面上与地域相关的因素做成异步方式获取

5.去掉 Cookie。在缓存的静态数据中不含有 Cookie。

有针对性地处理好系统的“热点数据”

热点操作

所谓“热点操作”，例如大量的刷新页面、大量的添加购物车、双十一零点大量的下单等都属于此类操作。

热点数据

“静态热点数据”

能够提前预测的热点数据。

“动态热点数据”

系统在运行过程中临时产生的热点。

发现热点数据

发现静态热点数据

实现方式是通过一个运营系统，把参加活动的商品数据进行打标，然后通过一个后台系统对这些热点商品进行预处理，如提前进行缓存。

发现动态热点数据

1 构建一个异步的系统，它可以收集交易链路上各个环节中的中间件产品的热点 Key

2 建立一个热点上报和可以按照需求订阅的热点服务的下发规范，主要目的是通过交易链路上各个系统（包括详情、购物车、交易、优惠、库存、物流等）访问的时间差.

3 将上游系统收集的热点数据发送到热点服务台，然后下游系统（如交易系统）就会知道哪些商品会被频繁调用，然后做热点保护。

流量削峰：削峰的存在，一是可以让服务端处理变得更加平稳，二是可以节省服务器的资源成本。

排队

消息队列（消峰，解耦，异步）来缓冲瞬时流量

答题

分层过滤

优化性能

线程数 = [(线程等待时间 + 线程 CPU 时间) / 线程 CPU 时间] × CPU 数量

减少cpu时间

1. 减少编码

2. 减少序列化

3.直接使用 Servlet 处理请求。避免使用传统的 MVC 框架

合适的线程数

减库存

下单减库存

解决大并发读问题

可以采用 LocalCache（即在秒杀系统的单机上缓存商品相关的数据）和对数据进行分层过滤的方式

写问题

把秒杀商品减库存直接放到缓存系统中实现

解决并发锁的问题

应用层做排队

按照商品维度设置队列顺序执行，这样能减少同一台机器对数据库同一行记录进行操作的并发度，同时也能控制单个商品占用数据库连接的数量，防止热点商品占用太多的数据库连接。

数据库层做排队

阿里的数据库团队开发了针对这种 MySQL 的 InnoDB 层上的补丁程序（patch），可以在数据库层上对单行记录做到并发排队。

付款减库存

预扣库存

保障稳定性

降级

当秒杀流量达到 5w/s 时，把成交记录的获取从展示 20 条降级到只展示 5 条。

限流：客户端和服务端限流是针对rpc调用来说的，发起方可以理解为客户端，调用方可以理解为服务端，限流就是分别限制发起方和调用方的次数

客户端限流

通过减少发出无用请求从而减少对系统的消耗。

服务端限流

根据服务端的性能设置合理的阈值

拒绝服务

高性能

高可用

一致性