前言

以下是我收集的一些问题，有的是网上摘录的，有的是自己参加面试被问到的，有的是工作或学习时遇到的，等等。

为什么要记录这些呢？

一方面，我相信，这样做对我自己的技术提升是有帮助的。在全文结构上我尽量**使问题连贯地形成知识体系**，而不是堆积的碎片，而且，每个问题我会尽量地给出答案。

另一方面，我希望，有大佬可以指出我的错误。因为我的答案不一定对，尤其那些带 * 的问题。

这份资料将会持续更新，如果有其他问题也可以留言讨论。欢迎交流，共同进步。

修改时间	版本	修改人	修改内容
2022-05-02	1.0.0	ZhangZiSheng001	初始化

JDK

以下问题主要针对 JDK 8 展开的。

JVM

class文件的组成结构

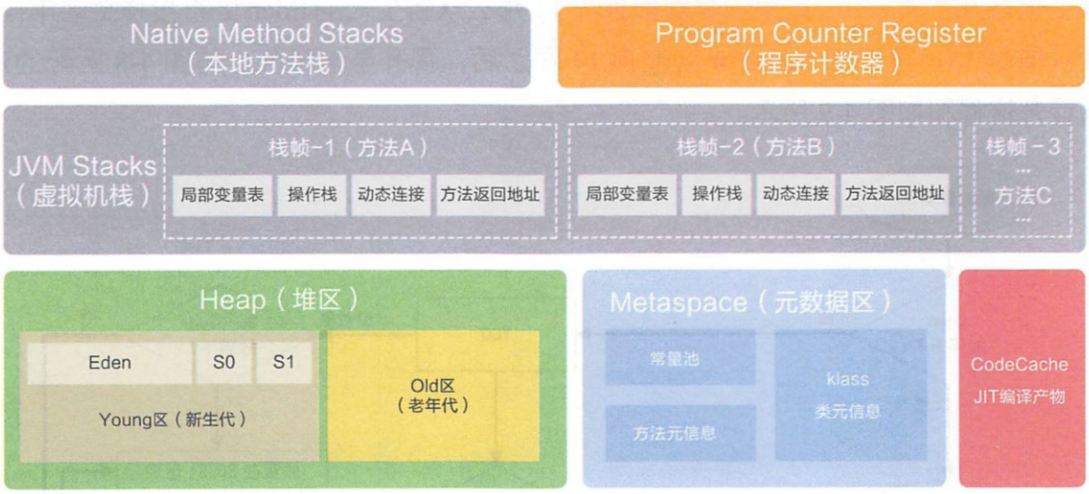
这个问题大致了解即可，它将帮助我们更好地认识 JVM 的运行时内存以及类的加载。具体可以看看这个系列的博客。

参考资料：Java虚拟机原理图解_亦山的博客-CSDN博客

说说JVM的运行时数据区

最低要求：我们需要知道有哪几个区？每个区具体存放什么东西？以及哪些区是线程共享？哪些区是线程私有？

更高要求：我们可以尝试在大脑里构建出一个运行中的 JVM，想象一下 new 一个新对象的过程，以及调用一个方法的过程。



图片来源：JVM内存模型看这个就够了技术交流牛客网 (nowcoder.com)

CompressedClassSpace是干嘛用的

我理解就是放 KClass 对象的空间，即 32 位指针 _compressed_klass 引用的那部分内存。

```
//hotspot/src/share/vm/oops/oop.hpp
class oopDesc {
private:
  volatile markOop _mark;
  union _metadata {
    Klass*      _klass;
    narrowKlass _compressed_klass;
  } _metadata;
}
//zzs0001
```

参考资料：深入探究JVM | klass-oop对象模型研究_sinolover的博客-CSDN博客

运行时常量池和字符串常量池的区别

运行时常量池：class 文件的静态常量池在运行时的表现形式，存放在方法区（元空间）。

字符串常量池：曾经属于运行时常量池的一部分，现已放入堆。

参考资料：运行时常量池和字符串常量池的区别

Minor GC、Major GC和Full GC的区别

Minor GC：新生代回收

Major GC：老年代回收

Full GC：JVM 报告里一般将它等价于 Major GC

一次完整的GC流程是怎样的

建议在大脑里想象一下 minor gc 和 major gc 的整个过程。minor gc 什么时候触发，是如何进行的？major gc 什么时候触发，又是如何进行的？

什么时候触发major gc

老年代空间不足。例如，空间分配担保失败、minor gc 后老年代放不下、大对象放不进老年代，等等

MetaSpace 空间不足

Compressed Class Space 空间不足

什么是分配担保机制

分配担保机制的存在是为了减少 gc 次数。

只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小，就会进行 Minor GC，否则将进行 Full GC。

对象什么时候晋升到老年代

Minor GC 时，eden + s0 的对象放不进 s1；

对象体积太大，放不进新生代；

Minor GC 时，age 到达阈值；

Minor GC 时，在 Survivor 区中相同年龄的对象的所有大小之和超过 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代

有哪几种垃圾回收算法

标记-清除算法

标记-复制算法

标记-整理算法

有哪几种垃圾回收器

我们需要知道有哪几种垃圾回收器？它们分别适用哪种场景？

你们项目用的哪一种？为什么？

Serial：用于新生代，单线程

ParNew：用于新生代，Serial 的多线程版本

Parallel Scavenge：用于新生代，吞吐量优先

Serial Old：用于老年代，单线程

Parallel Old：用于老年代，吞吐量优先

CMS：用于老年代，用户线程和回收线程可以并发执行。基于标记清除算法

G1：用于新老年代，用户线程和回收线程可以并发执行。

参考资料：JVM（HotSpot） 垃圾收集器 - 简书 (jianshu.com)

怎么手动触发gc

System.gc

jcmd PID GC.run

jconsole

新生代和老年代比例多大比较好*

默认情况下，young : old = 1 : 2。实际按项目运行情况做调整。

你们项目是怎么分配的?

字符串常量池会gc吗

会。因为放在堆里面。

MetaSpace会gc吗

会。只是它的垃圾回收比较特别，当 MetaSpace 或 Compressed Class Space 达到阈值，会触发堆的 full gc，回收堆时，顺便将堆中无效引用对应的 Class Metadata 等释放。

项目里用到的JVM参数

这里我给一个示例，我们需要知道每个参数的含义。

```
java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/my-service.hprof -
server -Xms4096m -Xmx4096m -Xmn1500m -XX:MetaspaceSize=256M -
XX:MaxMetaspaceSize=700M -XX:CompressedClassSpaceSize=64m -jar /home/my-
service.jar
```

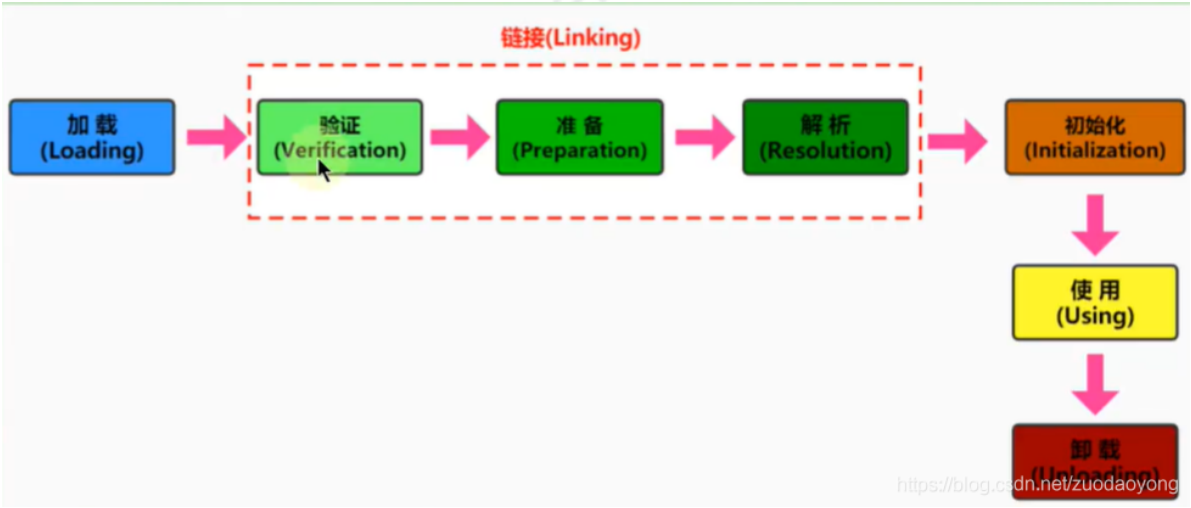
项目中JVM调优的例子*

运行时能不能调整堆大小*

应该是不行

类加载器

说说类的加载过程



参考资料：[类的加载过程详解zuodaoyong的博客-CSDN博客类加载]

什么是双亲委派机制

加载一个类时，先逐级检查父加载器是否已经加载，如果没有，再检查 BootstrapClassLoader 是否加载，如果还是没有，才会使用当前类加载器加载。

为什么要有多个类加载器

类库隔离。例如，tomcat 上运行多个项目

为什么会有双亲委派机制

复用某些类库，避免浪费内存

弥补多个类加载器存在的安全性

为什么破坏双亲委派机制

我觉得这是伪命题，根本没有破坏。所谓线程上下文类加载器并没有破坏双亲委派机制。

具体看看 SPI 的加载就知道了，`java.util.ServiceLoader.load(Class<S>)`，它只是将 Class.forName 用的类加载器改成线程上下文类加载器，线程上下文类加载器加载类的过程仍然遵循双亲委派机制。

```
    public static <S> ServiceLoader<S> load(Class<S> service) {
        ClassLoader cl = Thread.currentThread().getContextClassLoader();
        return ServiceLoader.load(service, cl);
    }// zzs001
```

网上各种说使用线程上下文类加载器就是破坏双亲委派机制，我不能理解。

Class.forName和ClassLoader.loadClass的区别

Class.forName：默认初始化静态成员和静态代码块，使用调用者的类加载器；

ClassLoader.loadClass：不初始化，使用当前类加载器。

自定义java.lang.String可以吗

可以，但你永远调用不到它。

JUC

juc 是面试中问到最多的，然而，我猜很多人和我一样，实际项目用到的比较少。不过，没关系，这些类库解决的问题不外乎这几个：

1. 如何使用多线程；
2. 多线程读写共享资源的安全问题；
3. 多线程顺序执行的问题；

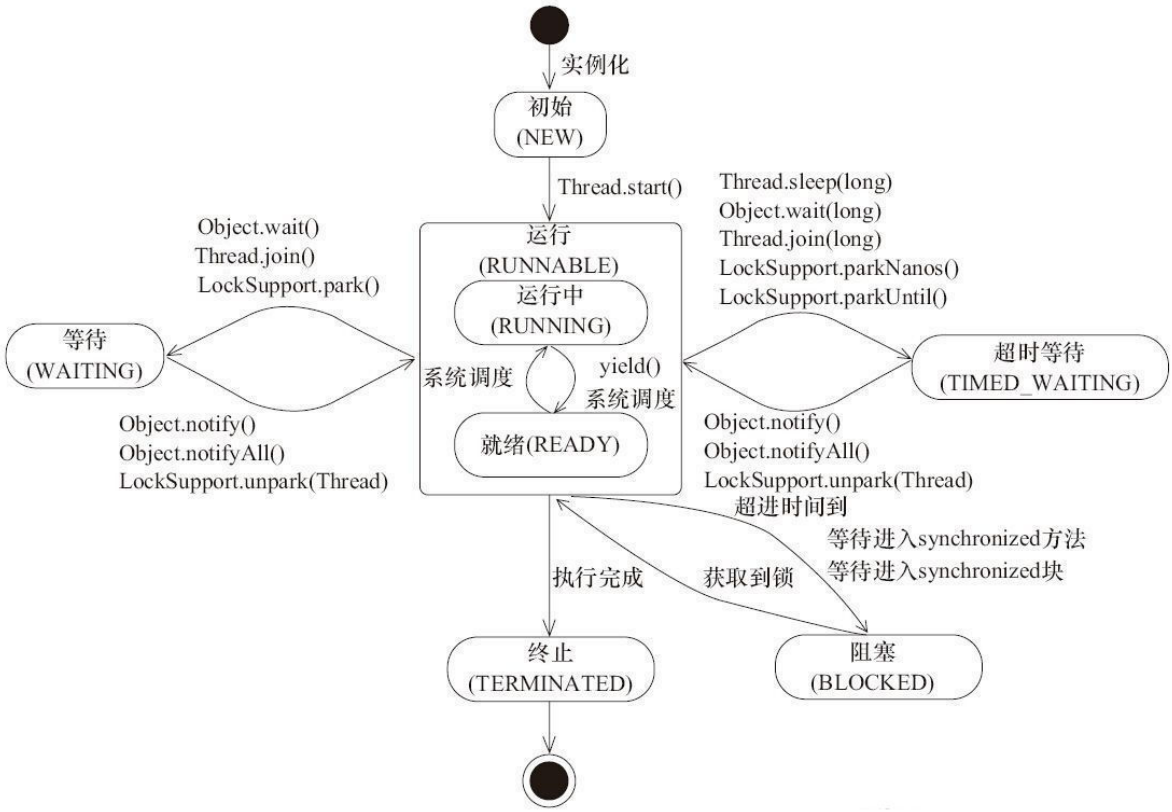
当我们尝试这么去划分的时候，学起来会简单很多。不要气馁，加油！！

线程和进程的区别

进程：资源分配的最小单位。每个进程使用独立的内存映射，进程间通信比较麻烦。

线程：程序执行的最小单位。一个进程下的所有线程共享同一内存映射，线程间通信比较方便。

线程有哪几个状态？它们如何转换的



图片来源：Java线程的6种状态及切换(透彻讲解)潘建南的博客-CSDN博客线程状态

如何中止一个线程

stop：直接停掉运行中的线程。不可控，不建议使用。

interrupt：只有被打断线程主动检查 interrupt 标识才有用，不然你 interrupt 也没用。

异步线程的异常怎么处理*

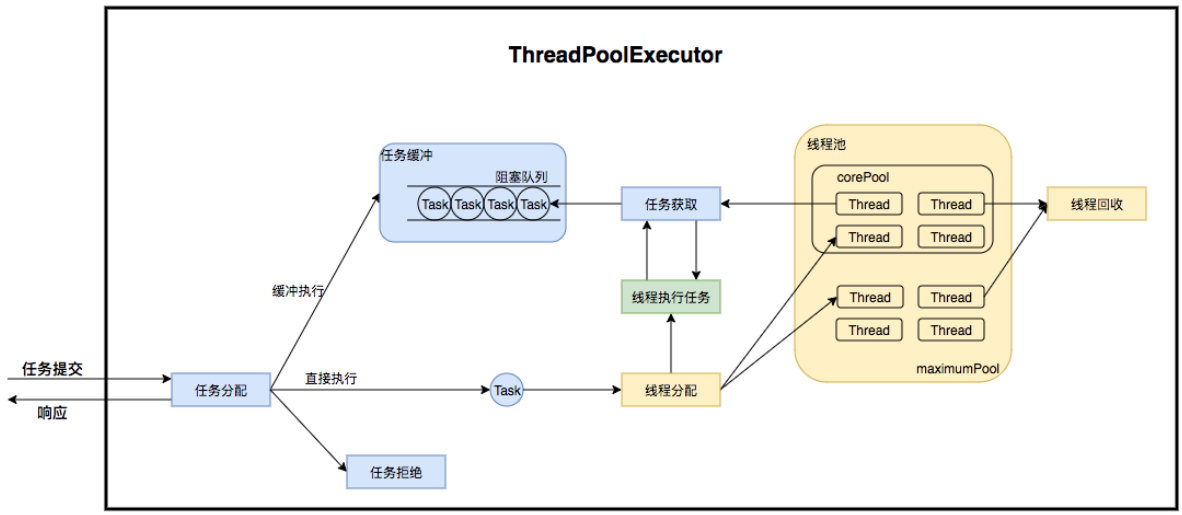
在 server 应用中我们经常会用全局异常处理器来处理各种异常，一般拦截的是 Controller，那异步线程的异常能捕获到吗？

如果不能，应该怎么处理？

如果不处理，异步线程的异常会写进日志吗？

什么是线程池？说说它的原理

美团团队的这篇文章几乎涵盖了线程池的大部分知识点，建议看看。我们需要重点了解线程池的整个结构以及运行机理。



参考资料：Java线程池实现原理及其在美团业务中的实践 - 美团技术团队 (meituan.com)

Executors的几种线程池

newCachedThreadPool：创建一个可缓存线程池，队列无容量。每进来一个新任务，如果没有空闲线程，都会新建线程。> coreSize 的空闲线程会 60s 后被回收；适用于前台任务。

newFixedThreadPool：创建一个定长线程池，队列无界。> coreSize 的空闲线程会被马上回收。适用于后台任务。

newScheduledThreadPool：创建一个定 coreSize 不定 maxSize 的线程池，队列无界、延迟，支持周期性任务执行。> coreSize 的空闲线程会被马上回收；适用于任务周期性执行的场景。

newSingleThreadExecutor：创建一个单线程化的线程池，队列无界，它只会用唯一的工作线程来执行任务。适用于一个任务一个任务执行的场景。

线程池的关闭方式

shutdown：停止接收新任务，已提交的任务继续执行。但调用完这个方法并不会阻塞等待任务执行完。

shutdownNow：停止接收新任务，还没执行的任务不执行，正在执行的任务 interrupt。

shutdown + awaitTermination：停止接收新任务，已提交的任务继续执行，阻塞等待任务执行完。

你们项目怎么用线程池*

一些需要快速响应的接口，多个事件异步处理；

定时任务消费，mq 消息消费；

tomcat 请求的处理；

等等

线程池设置多大合适*

线程池大小 = （（线程 IO time + 线程 CPU time ）/ 线程 CPU time ） * CPU数目

参考资料：Java并发线程池到底设置多大？ - Java架构师追风 - 博客园 (cnblogs.com)

ThreadLocal有什么用

在一个共享变量中存取各自线程的副本。建议看看源码。

你们项目中如何用的？？

使用ThreadLocal需要注意的问题

1. **脏数据。**线程复用时会拿到上一次的副本，使用完应及时 remove
2. **内存泄露。**ThreadMap 里 entry 的 key 为弱引用，gc 时会被回收导致大量 null = value 的 entry。使用完及时 remove

volatile的作用

保证可见性：读会读主存中的新值，写完会立即刷入主存。----一般读写的是基本变量

避免指令重排：例如，User u = new User()，会等对象初始化完再把引用给到 u。----双重检查锁的保障

什么是乐观锁、悲观锁

乐观锁：认为同时修改时小概率事件，所以不加锁，只是在修改时检查数据版本。例如，CAS、数据库版本号机制。

悲观锁：认为同时修改时大概率事件，所以加锁。例如，synchronized、数据库排它锁等。

你们项目中如何用的？？

CAS的作用

对资源进行原子性的读写操作，类似于 linux 的 TS 指令。可用于**实现乐观锁、竞争资源时快速失败、制造临界区**等。

针对并发不大或临界区逻辑简单的场景，效率比 synchronized 等要高。

CAS的缺点

1. **针对需要失败重试的场景，可能反复失败**

并发较大或事务逻辑较复杂时，CAS 会反复失败。

2. **ABA问题**

例如，取钱前 100 元，因为没反应所以连续点了 2 次取 50 元操作，结果先执行了一次，取出 50 元（100 -> 50），接着另一边有人给你存钱 50 元（50 -> 100），第二次取钱操作执行，又取出了 50 元（100 -> 50）。按理来说，第二次取钱操作应该失败才对。

解决办法：使用 AtomicStampedReference 和 AtomicMarkableReference 替代

3. **涉及多个变量时，无能为力**

解决办法：使用 AtomicReference 替代

什么是 AQS

AbstractQueuedSynchronizer，一个基础抽象类，可以用来实现独占锁、共享锁等同步器。

它维护了一个资源标识 state 和一个双向的 FIFO 线程等待队列，我们自定义同步器时只需要实现 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在顶层实现好了。

ReentrantLock、Semaphore、CountDownLatch、ReentrantReadWriteLock 都是基于 AQS 实现。建议都看看源码。

参考资料：Java并发之AQS详解 - waterystone - 博客园 (cnblogs.com)

什么是独占锁、共享锁

独占锁：同一时刻只有一个线程获得锁。例如，ReentrantLock

共享锁：同一时刻有指定数量的线程获得锁。例如，Semaphore

你们项目中如何用的？？

什么是公平锁、非公平锁

公平锁：先到先得。例如，new ReentrantLock(true)、Semaphore(permits, true)

非公平锁：先到不一定先得。例如，**synchronized**、new ReentrantLock(false)、Semaphore(permits, false)

你们项目中如何用的？？ zzs001

synchronized原理、优化

偏向锁：对象头的 MarkWord 存储持锁线程 id 并且锁状态标识为 01----只有一个线程时

轻量级锁：对象头的 MarkWord 存储栈帧的锁记录，基于 CAS----只要有两个线程就会升级轻量级锁

重量级锁：操作系统级别的锁，本质也是资源标识+同步队列----三个线程，或 cas 10 次失败就会升级重量级锁

可以看看 JDK1.6 对 synchronized 关键字的优化，随着锁竞争加剧，锁会升级且不可逆。

参考资料：Synchronized用法原理和锁优化升级过程(面试) - 叫练 - 博客园 (cnblogs.com)

CyclicBarrier与CountDownLatch区别

简单了解即可。

CountDownLatch 是一次性的，CyclicBarrier 是可循环利用的

CountDownLatch 是等待所有线程执行完成，CyclicBarrier 是等待所有线程都到位了再执行

某线程中断 CyclicBarrier 会抛出异常，避免了所有线程无限等待

三种等待/唤醒的使用和区别

Condition.await/signal：要求在 lock 的临界区内。await 时入队 condition queue，signal 时出队 condition queue 并入队 sync queue。

Object.wait/notify：要求在 synchronized 的临界区内。我觉得原理应该和 Condition.await/signal 差不多。

LockSupport.park/unpark：不要求在临界区内。LockSupport 唤醒时无需获得锁对象，而且可以唤醒指定线程。

你们项目用的哪一种？？

Object.notify和Condition.signal是随机唤醒线程吗

不是，FIFO 顺序唤醒。

建议自己写个例子测试。

如何保证多个线程顺序执行

join 这个不行，别听网上瞎说

newSingleThreadExecutor 这个不行，都说了多线程，别听网上瞎说

lock + Condition.await/signal

synchronized + Object.wait/notify

LockSupport.park/unpark

自旋判断式

用三个线程按顺序循环打印abc三个字母

根据上一个问题的答案，动手写

不sleep(0)和sleep(0)的区别

sleep(0) 虽然不会睡眠，但会让出 CPU，作用类似于 yield。

容器

说说常用的几个容器

这一部分的内容建议看看源码。

- Collection：存元素
  - List：元素可以重复，有序
    - **ArrayList**：底层数组。非线程安全。初始容量 10，一般按 oldCapacity + (oldCapacity >> 1)扩容（前提是插入的 index 要不超过 newCapacity），最大 Integer.MAX_VALUE。
    - LinkedList：底层双向链表。非线程安全。容量不预分配。也是双端队列
    - Vector：底层数组。线程安全。初始容量 10，一般按 oldCapacity + oldCapacity 扩容（前提是插入的 index 要不超过 newCapacity），最大 Integer.MAX_VALUE。
    - Stack：继承的 Vector，特征与 Vector 相同。
    - **CopyOnWriteArrayList**。底层数组。线程安全。容量不预分配。
  - Set：元素不能重复
    - HashSet：底层 HashMap
    - LinkedHashSet：底层 LinkedHashMap。
    - TreeSet：底层 TreeMap
  - Queue
    - ArrayDeque：底层数组。非线程安全。初始容量 16，一般按 oldCapacity + oldCapacity 扩容。
    - LinkedList：见上。
    - PriorityQueue：底层数组。非线程安全。初始容量 11，一般按 (oldCapacity < 64) ? (oldCapacity + 2) : (oldCapacity >> 1) 扩容
- Map：存 key-value 键值对

- **HashMap**：底层数组 + 链表/红黑树（链表长度大于 8，且数组长度大于 64 则会转换成红黑树）。非线程安全。初始容量 16，一般按 oldCapacity << 1 扩容。
- HashTable：底层数组 + 链表。线程安全。初始容量 11，一般按 oldCapacity << 1 扩容。
- LinkedHashMap：`HashMap` 和 `LinkedList` 的结合。特征与 HashMap 相同，只是它是有序的。可用于实现 LRU 缓存淘汰策略。
- TreeMap：底层红黑树。非线程安全。容量不预分配。
- **ConcurrentHashMap**：底层数组 + 链表/红黑树（链表长度大于 8，且数组长度大于 64 则会转换成红黑树）。线程安全。初始容量 16，一般按 oldCapacity << 1 扩容。

为什么HashMap的数组是2的n次方

位运算可以提高数据扩容和 key 取模效率

减小 hash 碰撞，元素分布更均匀

为什么链表大于8且size大于64才转红黑树*

红黑树是为了保证极端情况下的查找效率，缺点是空间大，增删代价大。应该在性能方面平衡好什么时候使用红黑树。

红黑树转回链表的阈值为什么是6不是8

因为如果这个阈值也设置成 8，假如发生碰撞，节点增减刚好在 8 附近，会发生链表和红黑树的不断转换，导致资源浪费

扩容因子为什么是0.75

空间成本和时间成本的平衡

HashMap和ConcurrentHashMap的区别

HashMap 线程不安全，允许 null key 和 null value

ConcurrentHashMap 线程安全，不允许 null key 和 null value

TreeMap的使用场景*

这个我暂时没遇到过需要使用 TreeMap 的场景。

ConcurrentHashMap为什么放弃分段锁

参考资料：[java8的ConcurrentHashMap为何放弃分段锁，为什么要使用CAS+Synchronized取代Segment+ReentrantLock - Lucky小黄人^_^ - 博客园 (cnblogs.com)]

DelayQueue的原理

简单了解即可。

DelayQueue 里维护一个 PriorityQueue，里面存放 Delayed 元素，元素按过期时间正序排列，每次取出时，会判断队首元素是否到期，如果没有阻塞或返回 null。具体建议看看源码。

Collections.synchronizedMap本质

装饰模式。注意获取的迭代器迭代时还是得手动加锁。

异常

Error和Exception的区别

Error：车发动机坏了，车子已经不能开了

Exception：车被拦住，车子还能开

参考资料：理解error和exception之间的区别 - hustzzl - 博客园 (cnblogs.com)

RuntimeException和其他Exception的区别*

可以从某一个方法的角度说，

RuntimeException：程序员的错？即这个方法写的不好

非 RuntimeException：调用方的错？即调用方传的参数有问题

你们项目中的业务异常是用的哪一种？为什么？我们项目所有业务异常类都继承了 RuntimeException，但这样做好像不大符合 RuntimeException 的定义，希望有大佬指点。

其他

String.length() 总是可靠吗

String.length() 来判断字符个数是否可靠?

不可靠。例如，"𪚥".length() = 2，String.length() 表示字符串由几个 char 组成，而有的字符需要两个 char 才能表示。可以用 codePointCount(int, int) 替代。

参考资料：为了彻底理解乱码问题，一怒之下我把字符集历史扒了个底朝天 - 双子孤狼 - 博客园 (cnblogs.com)

static方法可以重写吗

不能。子类可以声明同样方法名、同样出入参的方法，不过当你注解 @Override 时会编译报错，也就是说 static 方法可以重新声明，但不可以重写。

类的实例化顺序

- 父类静态成员和静态初始化块， 按在代码中出现的顺序依次执行
- 子类静态成员和静态初始化块， 按在代码中出现的顺序依次执行
- 父类实例成员和实例初始化块， 按在代码中出现的顺序依次执行
- 父类构造方法
- 子类实例成员和实例初始化块， 按在代码中出现的顺序依次执行
- 子类构造方法

redis

为什么要使用缓存

针对写少但读非常多的数据，尤其是数据组装比较复杂的，例如，字典、菜单树、部门树，等等，会考虑使用缓存来保护数据库和提升性能

为什么要使用redis

因为快

redis为什么快

数据读写为纯内存操作；

单线程；

高效的数据结构；

可以做很多事情。

redis为什么使用单线程

明确一点，通常说的 redis 单线程指的是读写数据使用单线程，而不是说整个 redis 进程就只有一条线程在运行。

我们可以先回答：为什么使用多线程？IO 指令时间约为 CPU 指令时间的 10^6 倍，当执行到 IO 指令阻塞了，应该让出 CPU 去执行其他进程的指令，本质就是为了提高 CPU 的利用率。

而 redis 读写数据是纯内存操作，并不包含 IO 指令，故无需采用多线程，使用单线程可以减少线程切换和锁竞争的开销。

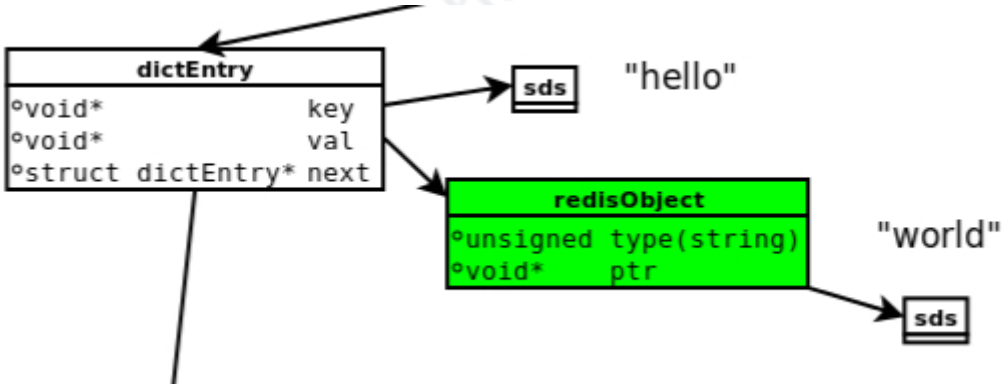
为什么redis6.0又用了多线程

redis 在处理客户端的请求时，包括获取 (socket 读)、解析、执行、内容返回 (socket 写) 等都由一个顺序串行的主线程处理，这就是早期的单线程模型。但是，这种模型存在一个缺点，就是不能发挥多核的能力。于是，redis 6.0 使用了多线程模型，不过，需要注意，Redis 的多线程模型只用来处理网络读写请求，对于数据读写，依然是单线程处理。

参考资料：[Redis 6.0 新特性：带你 100% 掌握多线程模型 - 码哥字节 - 博客园 (cnblogs.com)]

说说redis的数据结构

所有 key-value 都放在一张大的哈希表，存放 dictEntry 元素

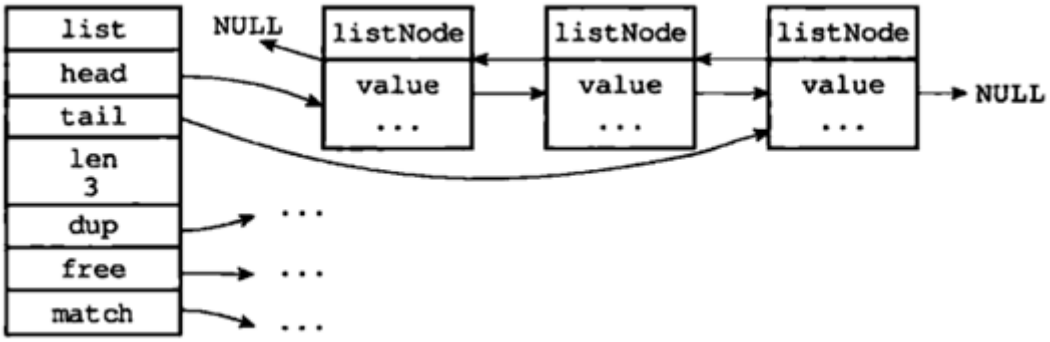


类 型	编 码	对 象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用 embstr 编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

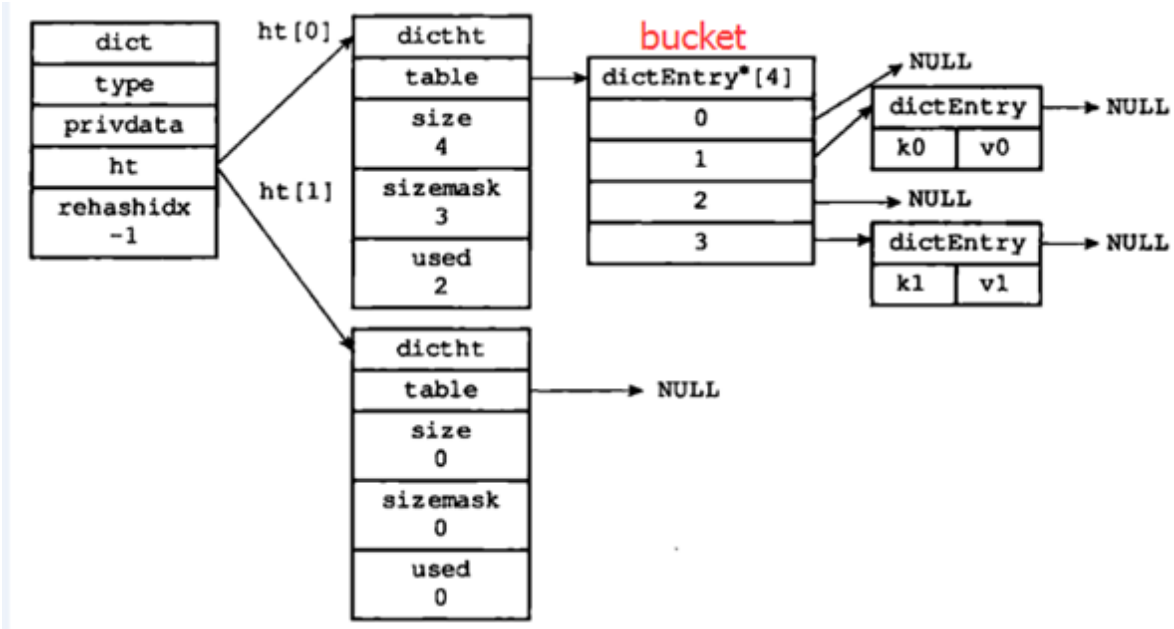
String：简单动态字符串 SDS

```
struct sdshdr {
    int len;
    int free;
    char buf[];
};
```

List：双向链表（元素数量小于512个且元素大小都不足64字节，才能为压缩列表）



Hash：哈希表（元素数量小于512个且元素大小都不足64字节，才能为压缩列表）



Set：哈希表（元素数量小于512个且元素都为整数，才能为整数数组）

```
// 整数数组
typedef struct intset{
    uint32_t encoding;
    uint32_t length;
    int8_t contents[];
} intset;
```

Sorted Set：基于单向链表的跳表（元素数量小于128个且元素大小都不足64字节，才能为可转压缩列表）

参考资料：深入学习Redis（1）：Redis内存模型 - 编程迷思 - 博客园 (cnblogs.com)

redis还可以做什么

string：（key-value）缓存部门树、字典树、共享 session；（增减）计数器、访问次数限制

list：消息队列

set：（随机）参与抽奖；（查找）IP 黑名单；（交集、并集、差集）共同关注、可能认识的人、商品筛选；（去重）商品标签？、点赞

zset：排行榜、延迟队列

hash：字典、分布式锁、购物车?

参考资料：一口气说出 Redis 16 个常见使用场景！ - bucaichenmou - 博客园 (cnblogs.com)

redis挂了怎么办

redis有哪种持久化

RDB：存快照数据。文件小，恢复快，对主进程性能影响较小，可能丢失 5 分钟的数据。

自动触发和手动触发

AOF：存写命令。文件大，恢复慢，对主进程性能影响较大，可能丢失更少数据，可读。

AOF 缓存区的同步文件策略 appendfsync 的正确理解（always、everysec、no）；

文件重写：自动触发和手动触发

使用 everysec 配置，AOF 最多可能丢失 2s 的数据，而不是 1s

混合持久化：4.0 开始引入。解决纯 aof 文件大恢复慢问题，解决纯 rdb 数据丢失较多问题。

aof-use-rdb-preamble yes

项目中使用哪种持久化？为什么

高性能，对数据丢失无所谓：不做持久化

高性能，对数据丢失较大容忍：RDB

对数据丢失难以容忍：混合持久化

怎么让redis不挂

说说redis的高可用架构

1. 主备模式

   完全复制

   部分复制：复制偏移量、复制积压缓冲区、服务器运行 ID(runid)

2. Sentinel 模式

   哨兵只是配置提供者，而不是代理

3. Cluster 模式

   MOD(CRC16(key)/2^14)

4. 其他

   主从读写分离 模式

   主从级联读写分离 模式

   多主模式？？

参考资料：Redis集群详解变成习惯-*CSDN*博客redis集群

你们项目怎么做的？为什么？Cluster 模式，不需要考虑主从不一致问题。

为什么Cluster模式不用一致性哈希，而是用哈希槽

一致性哈希存在数据倾斜、缓存击穿问题

参考资料：深度图解Redis Cluster原理 - detectiveHLH - 博客园 (cnblogs.com)

怎么解决一致性问题*

这个一致性包括主从读写一致性以及双写一致性。

我们只能保证最终一致性。一些对一致性要求比较高的场景，例如分布式锁和计数器，代码逻辑需要兼容不一致的情况。

redis的过期策略

定期过期（主动）

惰性过期（被动）

redis的淘汰策略

noeviction：新写入操作会报错

allkeys-lru：最近最少使用的 key 移除

allkeys-random：随机移除

volatile-lru：设置了过期时间的，最近最少使用的 key 移除

volatile-random：设置了过期时间的，随机移除

volatile-ttl：设置了过期时间的，更早过期的 key 移除

我们项目怎么做的？为什么？volatile-lru

LRU 和 LFU 的区别：LRU 最长时间未被使用，LFU 一定时间内被使用最少

使用时需要注意的问题

缓存穿透、缓存击穿、缓存雪崩、热key

缓存穿透：大量请求无 cache 且无 value。解决办法：缓存 null 值（单一 key 穿透）、key 校验、布隆过滤器（Redisson、Guava）

缓存击穿：大量请求无 cache 但有 value。解决办法：锁

缓存雪崩：大量 cache 集中过期。解决办法：随机过期时间、热键不过期、redis预热

超级热 key：有赞透明多级缓存解决方案（TMC）。客户端异步上报数据、流式计算系统热点探测、探测到热点通知客户端本地缓存。

如何解决双写不一致问题

其实，过期时间某种程度上也可以保证最终一致性

方案：

先更新数据库，再更新缓存：可能有脏数据。

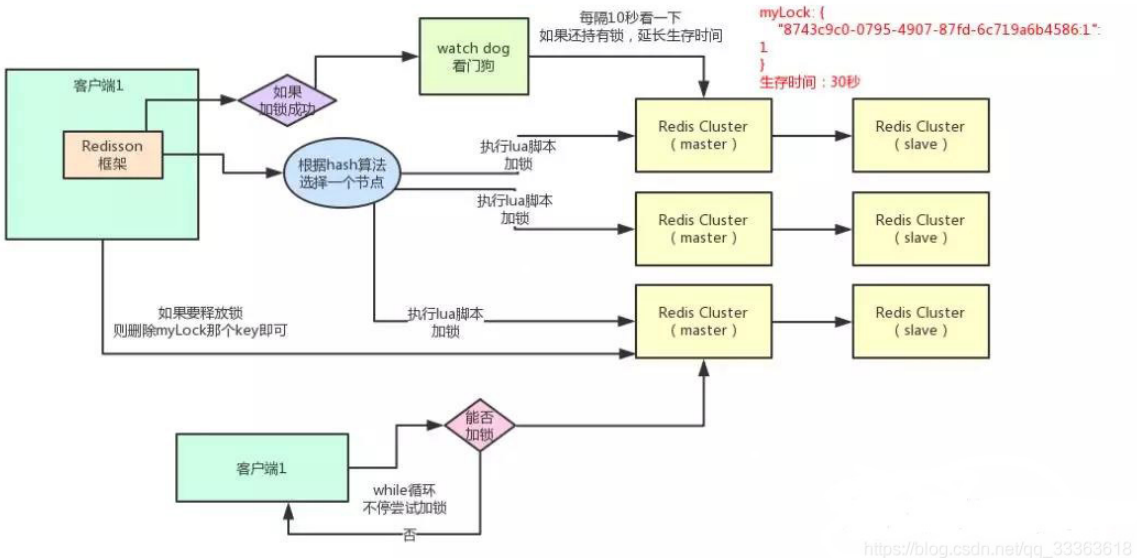
先删除缓存，再更新数据库：可能有脏数据（概率最大）。

先更新数据库，再删除缓存：可能有脏数据（概率最小）。

参考资料：[Redis与Mysql双写一致性方案解析 - 刘清政 - 博客园 (cnblogs.com)]

分布式锁

说说分布式锁原理



参考资料：[分布式锁-Redission彩色天的博客-CSDN博客redission]

如何避免多客户端获得同一把锁

避免多个客户端拿到同一把锁：

防止宕机导致的一锁多持：红锁，代价非常大

防止超时导致的一锁多持：定时延迟锁

最坏的打算，多个客户端拿到同一把锁是必然存在的：只能解自己的锁 + 解锁失败回滚业务逻辑

其他

超时时间大小设置的依据是什么*

如何实现一个列表中元素不同过期时间

用 zset 存列表，score 为过期时刻，取数时用 zrangebyscore。

mysql

为什么要用数据库

数据需要持久化以及检索

为什么要用mysql*

关系型数据库和非关系型的区别

扩展性：数据的结构是否写死的、表与表之间是否关联、约束关系

性能、数据丢失：内存和硬盘

事务支持：数据一致性的要求

SQL和NOSQL怎么选择

做持久化、数据操作复杂：SQL

高性能且容忍数据丢失、数据操作简单：NOSQL

读写的高效

为什么查询那么快

基于 B+ 树的索引结构

什么是B+树

【查找结构1】静态查找结构概论 - 爪哇人 - ITeye博客

【查找结构 2】二叉查找树 [BST] - 爪哇人 - ITeye博客

【查找结构3】平衡二叉查找树 [AVL] - 爪哇人 - ITeye博客

【查找结构5】多路查找树/B~树/B+树 - 爪哇人 - ITeye博客

【查找结构4】红黑树 [RBT] - 爪哇人 - ITeye博客

【查找结构6】动态查找树比较 - 爪哇人 - ITeye博客

数组折半查找 O(logN)，增删代价太大，于是

二叉树查找 O(logN)，最坏时间复杂度 O(N)，于是

平衡二叉树查找 O(logN)，增删代价较大，于是

红黑树查找 O(logN)，大量数据存储中由于树深度过大而造成磁盘 IO 读写过于频繁，于是

B 树查找 O(logN)，进一步减少 IO，于是

B+ 树查找 O(logN)

hash索引和B+树索引的区别

等值查询的稳定性

是否有序、范围查询是否支持

聚簇索引和非聚簇索引的区别

聚簇索引：一般为主键索引

非聚簇索引：一般为主键索引之外的索引

组合索引生效规则

参考资料：mysql联合索引的生效规则 - mintsd - 博客园 (cnblogs.com)

复合索引的索引树是怎样的*

例如，复合索引列 a、b、c，我觉得它的树大致是这样的：a 的索引树的叶子节点是 b 的索引树，b 的索引树的叶子节点是 c 的索引树，而 c 的索引树的叶子节点是主键的索引树。

怎么判断索引是否生效

explain 一下，看看对应的 type 和使用的索引。

system > const > eq_ref > ref > range > index > all

system：该表只有一行

const：唯一索引等值查询

eq_ref：联表时唯一索引等值查询

ref：非唯一索引等值查询

range：索引范围查询

index：扫描整个索引树

all：扫描全表

重复性高的字段都不能加索引吗

如果查的是多数，加索引没必要，如果查的是少数，加索引速度快很多。

这个问题主要是纠正一个看法，就重复性非常高的字段一定不能加索引。这个嘛，还是要看场景。

项目中有没遇到慢查询？怎么办

一般是 explain 一下，看看有没有用上索引，尽量优化为使用索引或更优的 type，实在不行考虑给字段加索引或者代码层面优化。

sql的执行顺序

参考资料：sql执行顺序 - qanholas - 博客园 (cnblogs.com)

exists和in的区别

1: select * from A where exists (select * from B where B.id = A.id); query 循环次数为 A 表的记录数，用到 B 表索引；

2: select * from A where A.id in (select id from B);query 循环次数为 B 表记录数，用到 A 表索引。

3:select * from A where not exists (select * from B where B.id = A.id);query 循环次数为 A 表的记录数，用到 B 表索引；

4:select * from A where A.id not in (select id from B);query 循环次数为 B 表记录数，不会用到索引。

无论那个表大，用not exists都比not in要快。

count(*)、count(字段) 和count(1)区别

count(字段) < count(主键id) < count(1) ≈ count(*)

参考资料：mysql中count()、*count(1)*和*count(*字段)的区别桐花思雨的博客-*CSDN*博客*mysql*中*count1*和*count*的区别

为什么select count(*)在myisam比Innodb快

MVCC 的影响

事务、锁

什么是事务

类似于 synchronized、lock 锁住的临界区，只是 mysql 的事务更加复杂，因为要并发读写，而且锁住哪些资源不能预先知道。

脏读、不可重复读、幻读的原因和解决方案

脏读：还没结束的写操作被读操作分割了。解决方案：给读加锁或将事务隔离级别更改为读已提交；

不可重复读：还没结束的读操作被写操作分割了。解决方案：给读加锁或将事务隔离级别更改为可重复读；

幻读：还没结束的读写操作被插入操作分割了。解决方案：将事务隔离级别更改为可重复读（只能解决一部分）；

什么是MVCC

参考资料：MySQL笔记五-MVCC - 镰鼬LL - 博客园 (cnblogs.com)

RR级别能否解决幻读

如果是读-读的幻读，那 RR 级别可以解决幻读；--原理MVCC

如果是读-写的幻读，那 RR 级别不能解决幻读，但可以在 RR 级别下给读加锁解决。--原理 next-key lock

gap锁和next-key锁的关系

next-key lock，就是 record lock 和 gap lock 的结合，即除了锁住索引本身，还要再锁住索引之间的间隙。

next-key的一些示例

已知表：id 为主键索引，num 为普通索引

id	num	desc
5	5	5
10	10	10
15	15	15

以下 sql 加的锁：

sql	加的锁
select * from gap_table where id = 10 for update;	id=10的record锁
select * from gap_table where num = 10 for update;	num in (5,15)的next-key锁
select * from gap_table where id > 10 for update;	id in (10,+∞)的next-key锁
select * from gap_table where num > 10 for update;	num in (10,+∞)的next-key锁
select * from gap_table where id = 12 for update;	id in (10,15)的next-key锁
select * from gap_table where num = 12 for update;	num in (10,15)的next-key锁
select * from gap_table where id > 12 for update;	id in (10,+∞))的next-key锁
select * from gap_table where num > 12 for update;	num in (10,+∞))的next-key锁

为什么要有意向锁

参考资料：InnoDB 的意向锁有什么作用？ - 知乎 (zhihu.com)

MyISAM 和 InnoDB 的区别

是否支持事务

是否支持行级锁

如何解决死锁*

1. 等待，直到超时（innodb_lock_wait_timeout=50s）。
2. 发起死锁检测，主动回滚一条事务，让其他事务继续执行（innodb_deadlock_detect=on）

参考资料：MySQL如何处理死锁 - fdbnf - 博客园 (cnblogs.com)

MVCC下delete的数据会一直存在吗

不会。因为会定期 purge 掉。

高可用

为什么分库分表

单库数据量过大或请求量过大

单表数据量过大或请求量过大

怎么分库分表

单库数据量过大或请求量过大：分库。按业务模块拆分

单表数据量过大或请求量过大：分表。垂直拆分，分列；水平拆分，分行，每 500w 一张表

有哪些分库分表方案

sharding-jdbc：client层方案--中小型公司

参考资料：sharding-jdbc 分库分表的 4种分片策略，还蛮简单的 - 程序员内点事 - 博客园 (cnblogs.com)

mycat：proxy层方案--中大型公司

mysql的主从复制方式

强同步

半同步

异步

mysql的高可用方案*

mysql 自带

- MySQL Replication：一主一备
- MySQL Fabirc：一主一备 + 一主多从 + 分片
- 参考资料：MySQLFabric概述 - 华行天下 - 博客园 (cnblogs.com)
- MySQL Cluster：？
- 参考资料：实战体验几种MySQL Cluster方案（转） - 庞国明 - 博客园 (cnblogs.com)

mysql第三方优化

- MMM：双主多从，我理解就是一主一备多从。
- MHA：一主多从
- Galera Cluster：多主

依托硬件配合

- heartbeat+SAN
- heartbeat+DRDB

其它

Zookeeper + proxy

Paxos

你们项目怎么做的？为什么？阿里云的集群版：一主一备多从

主从不一致怎么解决

我们只能保证最终一致性。一些对一致性要求比较高的场景，代码逻辑需要兼容不一致的情况。

如果无法容忍主从不一致，应该选择强一致的架构。

其他

canal*

队列

为什么使用队列

解耦：事件的发布订阅，我不需要知道有谁在关注我的事件

异步：为了更快的响应

削峰：平滑请求

进程间通信

分布式事务

参考资料：为什么使用消息队列?消息对列有什么好处? - 爱笑的Terry - 博客园 (cnblogs.com)

为什么要用RabbitMQ*

项目里怎么用的

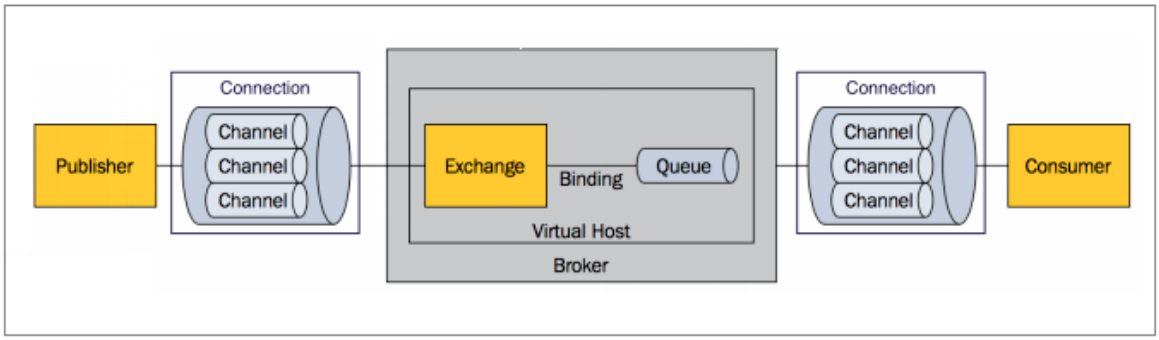
订阅制造系统的订单进度：解耦

将订单传给制造系统：进程间通信

一键下单快速响应：异步

RabbitMQ的架构

AMQP的协议模型



RabbitMQ的两种模式

只有队列

交换机 + 队列：四种交换机

参考资料：RabbitMQ详解（三）------RabbitMQ的五种队列 - YSOcean - 博客园 (cnblogs.com)

RabbitMQ的四种交换机

我觉得本质区别就在于：你打算把不同类型的消息发到一个 exchange 还是多个 exchange。

direct：

topic：

fanout：

headers：

routingkey和bindingkey的区别

routingkey 根据 bindingkey 来决定发送到哪个队列。

需要考虑的问题

如何保证消息有序*

MQ 的消息本身就是有序的，之所以会出现无序，在于一个队列存在多个消费者，而这些消费者处理消息的顺序是不可控的。

最简单的方法就是，只让一个线程来消费，在定义 queue 时设置参数 x-single-active-consumer = true，以及在定义消费端 RabbitListenerContainerFactory 时设置 perfetch = 1。

如何保证消息不丢失

1. MQ：exchange、queue、message 需要持久化

   exchange：durable = true and autoDelete = false

   queue：durable = true and autoDelete = false

   message：deliveryMode = 2

2. 发送：

消息记录入库，发送成功更新记录，失败定期重试（针对有 exchange 无 queue 的情况，发送端不需要关注，alternate-exchange = DLX ）

3. 消费

手动消费确认 + 重复消费处理

参考资料：RabbitMQ最佳实践 - 无知者云 - 博客园 (cnblogs.com)

怎么保证消费幂等性

消费逻辑本身幂等

消费成功记录入库

如何避免队列消息积压

什么原因导致的堆积?

1. Consumer 消费慢了----优化消费逻辑或扩容 Consumer
2. Producer 发快了：突发流量----扩容 Consumer或降级 Producer
3. 消息重复消费失败----丢进死信队列

如何做延迟队列

TTL + DLQ

重回队列的消息排在哪里

队首

高可用

RabbitMQ的高可用方案

普通集群：类似分片，但没有考虑容灾。

镜像集群：多主架构

思考：可否将以上两种模式结合呢？即普通集群里每个节点做主备

我们项目怎么做的？为什么？镜像集群 + HAProxy：3 个节点

参考资料：RabbitMQ高可用原理讲解 - 简书 (jianshu.com)

其他

生产者创建队列还是消费者创建队列

如果让生产者创建：生产者需要预先知道有哪些队列，新增队列需要重新配置和部署生产者，不够解耦；

如果让消费者创建：消费者必须知道它所绑定的交换机，交换机的任何更改都需要重新配置和部署所有消费者，不够解耦。

优化：可以使用管理界面或脚本创建交换机、队列以及它们之间的关系，这个时候，生产者只需要知道交换机，消费者只需要知道队列。

死信交换（DLX）和死信队列（DLQ）

设置 x-dead-letter-exchange 为 DLX，在以下三种情况下消息将被扔到DLX中：

1. 消费方 nack 时指定了 `requeue=false`
2. 消息的 TTL 已到
3. 消息队列的 max-length 已到

推模式和拉模式

spring

IOC和DI的理解

IOC 和 DI 应该说的是同一个事情，就是让容器帮我们注入依赖，而不是我们自己注入。

beanFactory和applicationContext的区别

applicationContext 继承了 beanFactory，有它的所有功能，还提供了更多的功能；

beanFactory 中，很多功能需要以编程的方式实现，比较面向 spring 内部，而在 applicationContext 中则可以通过配置实现，更面向使用者；

beanFactory 要 getBean 时才构建 bean，而 applicationContext 预先构建 bean；

如何解决循环依赖

通过将实例化后的对象提前暴露给 Spring 容器中的 singletonFactories，解决了循环依赖的问题

参考资料：Spring中的循环依赖解决详解 - 淡墨痕 - 博客园 (cnblogs.com)

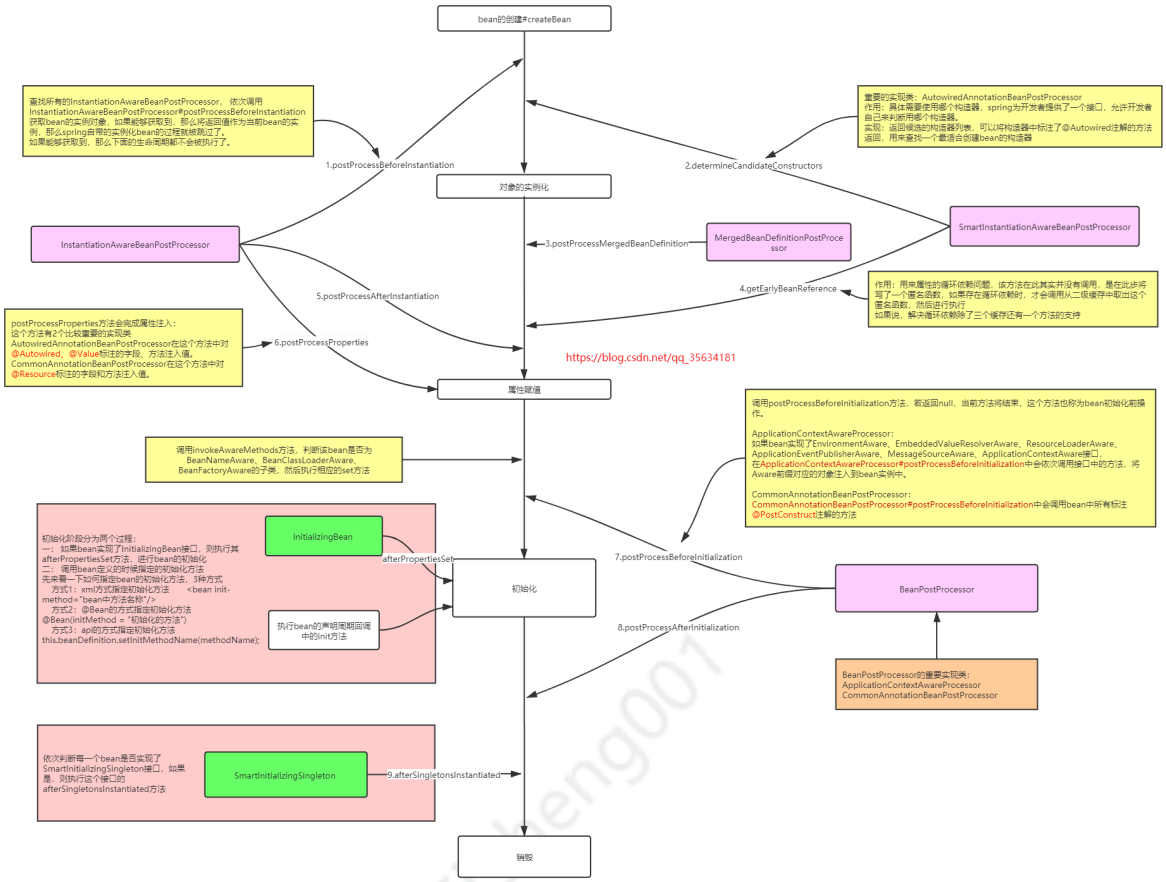
什么情况会出现无法解析的循环依赖？为什么

1. userService 和 userDao 相互依赖，且它们均为多例
2. userService 为单例，userDao 为多例。如果你先获取的是多例的 userDao，则会报错；如果你先获取的是单例的 userService，那么不会报错

参考资料：Spring源码系列(一)--详细介绍bean组件 - 子月生 - 博客园 (cnblogs.com)

bean的生命周期

这个图比较直观一些，不过还是少了 postProcessorsBeforeInstantiation postProcessAfterInstantiation、postProcessProperties



图片来源：Spring中bean的生命周期（最详细）*goodluckwj*的博客-*CSDN*博客bean生命周期

事务传播机制

调用方	无事务时非事务	无事务时抛出异常	无事务时新建事务
有事务时加入	PROPAGATION_SUPPORTS	PROPAGATION_MANDATORY	PROPAGATION_REQUIRED
有事务时新建事务			PROPAGATION_REQUIRES_NEW
有事务时抛出异常	PROPAGATION_NEVER		
有事务时嵌套事务			PROPAGATION_NESTED
有事务时非事务	PROPAGATION_NOT_SUPPORTED		

PROPAGATION_REQUIRED：如果当前存在事务，就加入该事务，如果当前没有事务，就创建一个新事务；

PROPAGATION_SUPPORTS：如果当前存在事务，就加入该事务，如果当前没有事务，就以非事务执行；

PROPAGATION_MANDATORY：如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常；

PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务；--外的异常不会让内回滚

PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起；

PROPAGATION_NEVER：以非事务方式执行操作，如果当前存在事务，则抛出异常；

PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。--与PROPAGATION_REQUIRES_NEW相比，外的异常会让内回滚，与PROPAGATION_REQUIRED相比，内的异常可以catch掉从而不影响外

你们项目用过哪几种？？

Spring Cloud中feign、ribbon和hystrix的超时时间

feign：默认 connectTimeout 10 s，readTimeout 60 s。

ribbon：默认 connectTimeout 1 s，readTimeout 1 s。

hystrix：默认 executionTimeout 1 s。

如果都没有配置，优先 ribbon 的默认配置；如果都配置了超时，feign 优先于 ribbon；

hystrix 的超时设置：

(1 + MaxAutoRetries) * (1 + MaxAutoRetriesNextServer) * ribbon ReadTimeOut < hystrix 的 executionTimeout

springboot自动装配的过程

参考资料：你来说一下springboot启动时的一个自动装配过程吧 - 纪莫 - 博客园 (cnblogs.com)

微服务

为什么要用微服务

我们希望应用不断壮大的同时，仍能提供较好的用户体验。单体应用无法通过横向扩展解决性能瓶颈，需要纵向拆分。

另外，如此庞大的单体应用，可维护性和可靠性较差。

为什么要使用集群而不是更好的单机

价格有效性、可伸缩性、高可用性

什么是CAP

这里我说说自己的理解，可能和网上大部分解释不一样。P 是某个机器挂了，整个服务不影响，像主备容灾就是 P；A 是更好的性能，例如集群就是 A；C 是一致性。

当然，你也可以看看普遍的解释：

参考资料：CAP理论的理解 - John_nok - 博客园 (cnblogs.com)

什么是Service Mesh*

参考资料：微服务架构基础之Service Mesh - tianyamoon - 博客园 (cnblogs.com)

什么是降级

开关降级、限流降级、熔断降级

几种限流算法

计数器固定窗口：

计数器滑动窗口：

漏桶：不支持突发流量，真的如此吗？？

令牌桶：

参考资料：常用限流算法 - 香芋牛奶面包 - 博客园 (cnblogs.com)

分布式事务*

两种场景：单个应用，但是涉及多个数据存储；多个应用，每个应用可能连接着一个或者多个数据存储

2PC：强一致性

3PC：强一致性

TCC：强一致性

本地消息表：最终一致性

你们项目怎么用的？？

分布式id

uuid

数据库

redis

雪花

oauth2.0

授权码模式

token 模式

账户密码模式

客户端模式

eureka、ribbon、hystrix是如何配合的*

你们项目的日活、qps是多少

PV：

UV：

QPS：

怎么统计qps

1. skywalking
2. SLB
3. tomcat、网关日志的 access.log

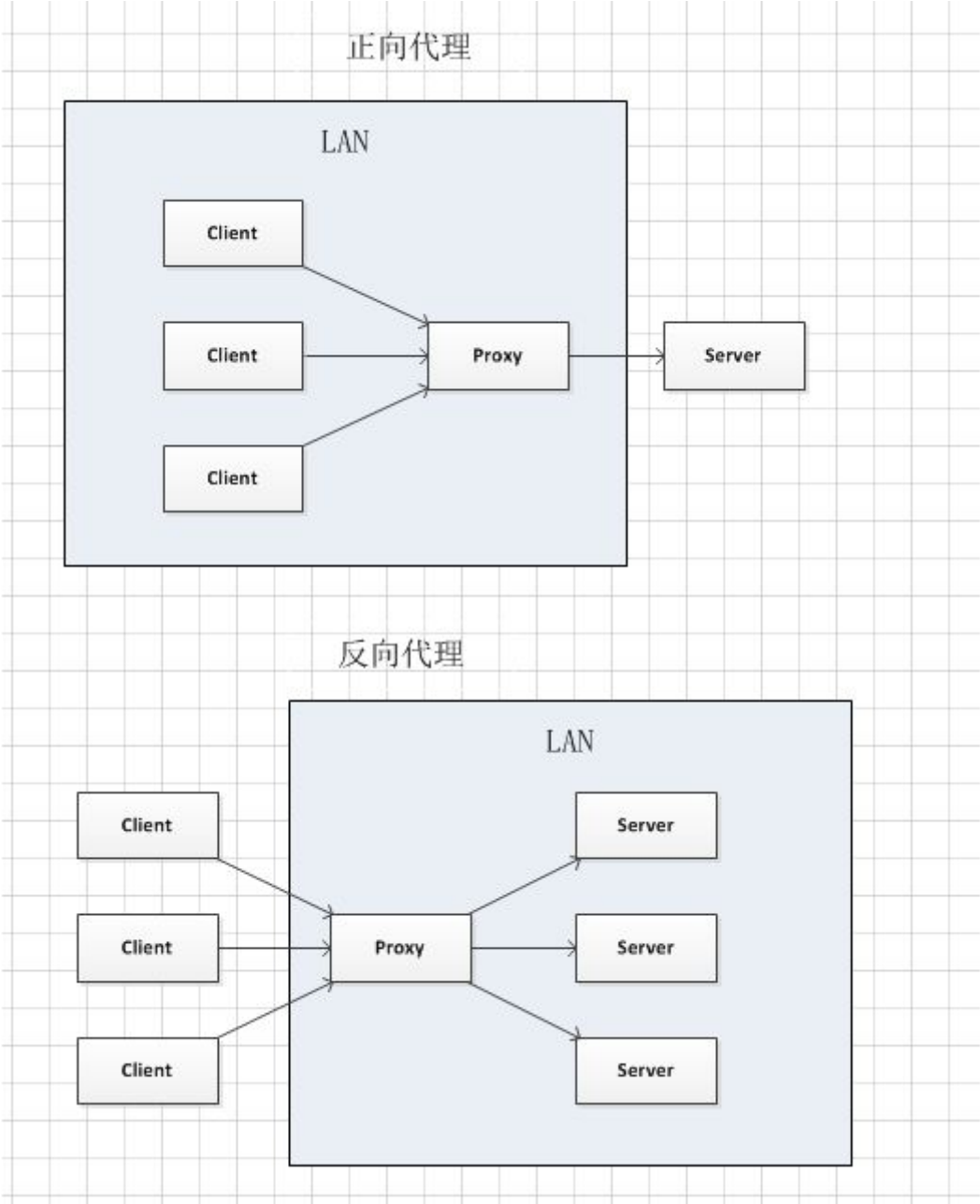
```
cat access.log |grep mtdsretail|cut -d ' ' -f2|cut -c 1-8|uniq -c|sort -n -r
```

3. 用 PV 估算：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

正向代理和反向代理的区别

正向代理即是客户端代理，代理客户端，服务端不知道实际发起请求的客户端。例如，VPN

反向代理即是服务端代理，代理服务端， 客户端不知道实际提供服务的服务端。例如，网关、负载均衡器



参考资料：反向代理和正向代理区别 - 泛夜泰克 - 博客园 (cnblogs.com)

如何实现热部署*

你们项目的监控告警

Prometheus + Grafana

参考资料：14. Prometheus 快速入门教程 - 随笔分类 - 陈树义 - 博客园 (cnblogs.com)

Counter、Gauge、Summary、Histogram的区别

Counter：计数器。例如请求次数

Gauge：常规数值。例如内存占用率

Histogram：分区间统计。

Summary：按比例统计。

链路

skywalking

操作系统

Linux的IO模型有哪几种

阻塞IO（bloking IO）：等待数据阻塞

非阻塞IO（non-blocking IO）：等待数据轮询，不需要阻塞

多路复用IO（multiplexing IO）：等待数据阻塞轮询多个 IO，需要阻塞

信号驱动式IO（signal-driven IO）：AIO

异步IO（asynchronous IO）：AIO，复制数据不需要我处理

参考资料：聊聊Linux 五种IO模型 - 简书 (jianshu.com)

reactor模型有哪几种

单 reactor 单线程：无法利用多核能力

单 reactor 多线程

多 reactor 多线程

参考资料：【NIO系列】——之Reactor模型 - wier的个人空间 - OSCHINA - 中文开源技术交流社区

epoll和poll的区别

select 和 poll 在“醒着”的时候要遍历整个 fd 集合，而 epoll 在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的 CPU时间。

参考资料：linux的五种IO模型 - lovejune - 博客园 (cnblogs.com)

平时用到的Linux命令

grep -C

tail -f

less -N

scp

top -p

ps -ef

一个进程最多可以创建多少线程

参考资料：一个进程最多可以创建多少个线程？ - 知乎 (zhihu.com)

故障排查

为什么 java 程序的内存溢出往往伴随着 CPU 爆表

java 内存溢出一般会伴随堆中大量对象无法回收，这个时候会频繁地 gc，而 gc 执行的主要是 cpu 指令（而非 IO 指令），故 cpu 占用率高。

CPU爆表怎么排查

arthas：

dashboard：获取 cpu 占用率最高的线程

thread pid：获取该线程堆栈

jad className：反编译该类

jdk：

top：查找出哪个进程消耗的 cpu 高

top -p ： 查看具体进程，按 shift+h 或“H”，打开线程视图

printf '%x\n' ： 将线程号转为16进制

jstack |grep -A 100 0x：查看堆栈

内存高怎么排查

看监控，eden、old、metaspace、stack 的内存分布；或 jmap -heap

dump 文件查看

线上系统突然变得异常缓慢，如何排查*

单个接口：日志、链路

大批量：日志、CPU、内存、磁盘、网络

网络

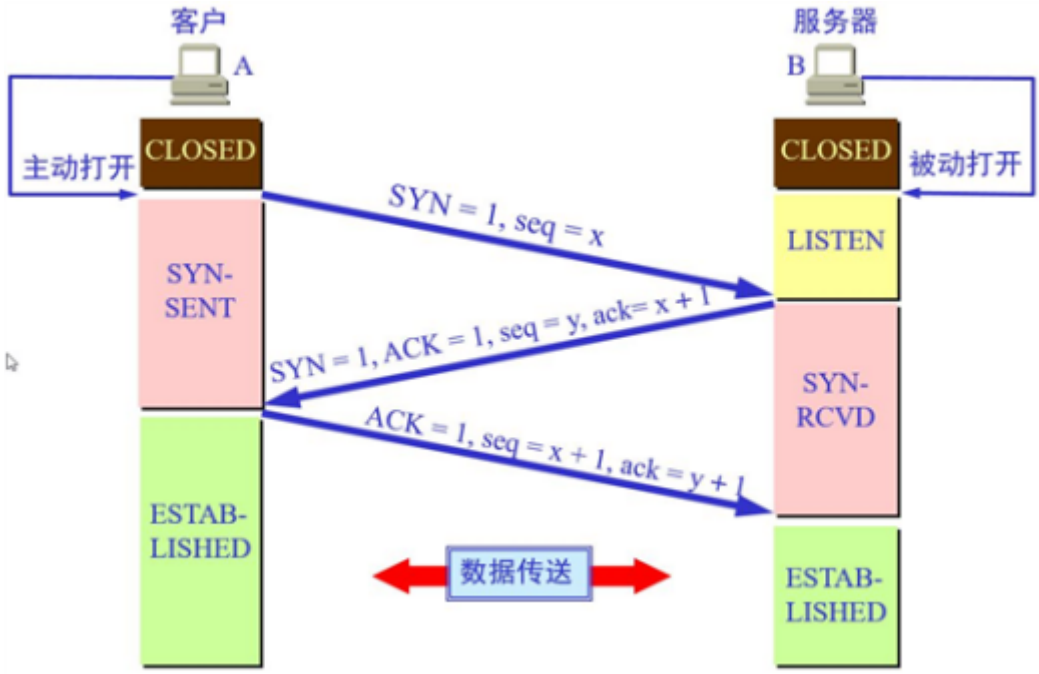
网络：集线器、交换机、路由器

这个问题可以略过。就是我无意中看到一篇好文章，就当问题做个记录。

> 按照发现问题、解决问题的思路，我捋了一下：
>
> 首先，电脑A直接连接其他电脑来通信，随着通信伙伴增多，存在问题：端口不够用以及连线复杂。
> 为了解决连线复杂，有了集线器。但遇到新的问题：群发导致的信息不安全和资源浪费。
> 为了集线器的问题，有了能定向通信的交换机。但端口不够用问题仍然存在。
> 为了端口不够用，有了串联的交换机。但又遇到新的问题：mac地址映射表过于庞大。
> 为了解决映射表庞大的问题，我们考虑将子网以外的数据全部转发到另外一台机器，于是有了路由器。
> ......

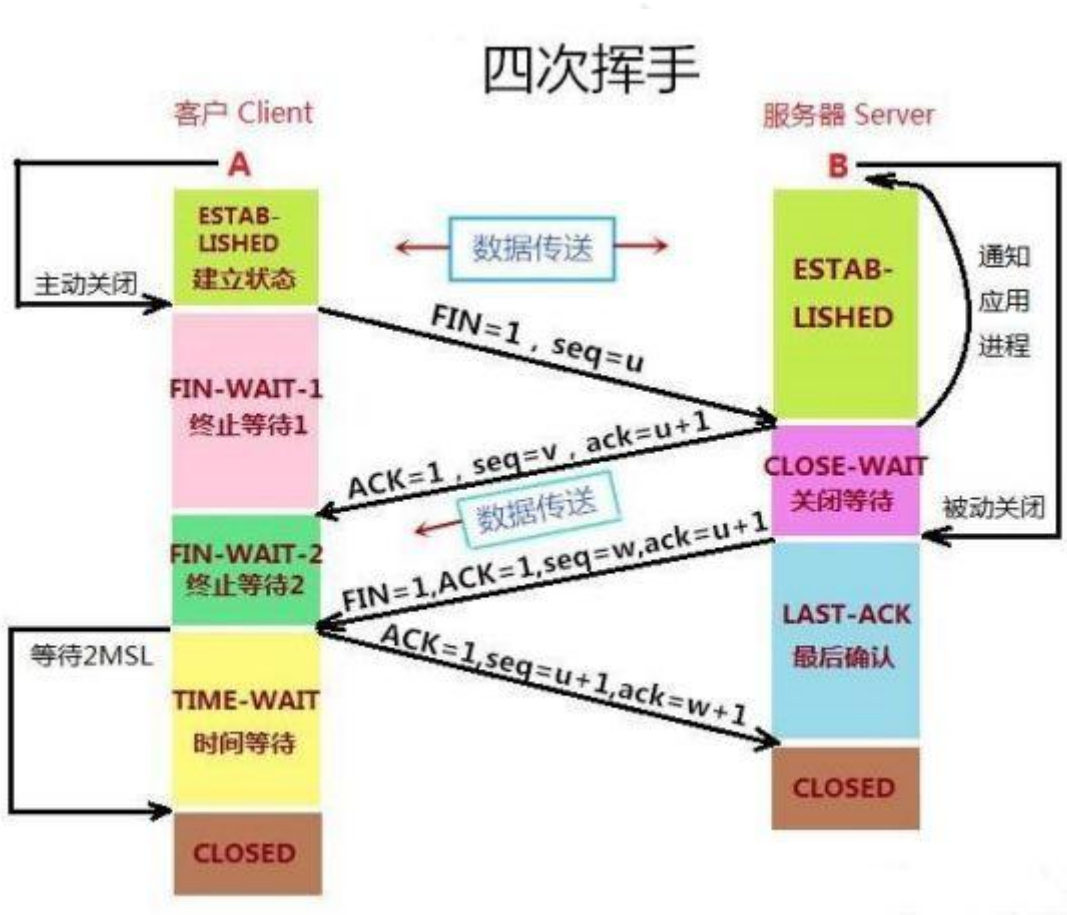
参考资料：图解 | 原来这就是网络 - 闪客sun - 博客园 (cnblogs.com)

为什么是三次握手？最后一次省略不行吗



图片来源：TCP协议之三次握手、四次挥手 - 简书 (jianshu.com)

为什么是四次挥手？第二次和第三次合并不行吗？最后一次省略不行吗



图片来源：TCP协议之三次握手、四次挥手 - 简书 (jianshu.com)

TCP四次挥手为什么要等待2msl

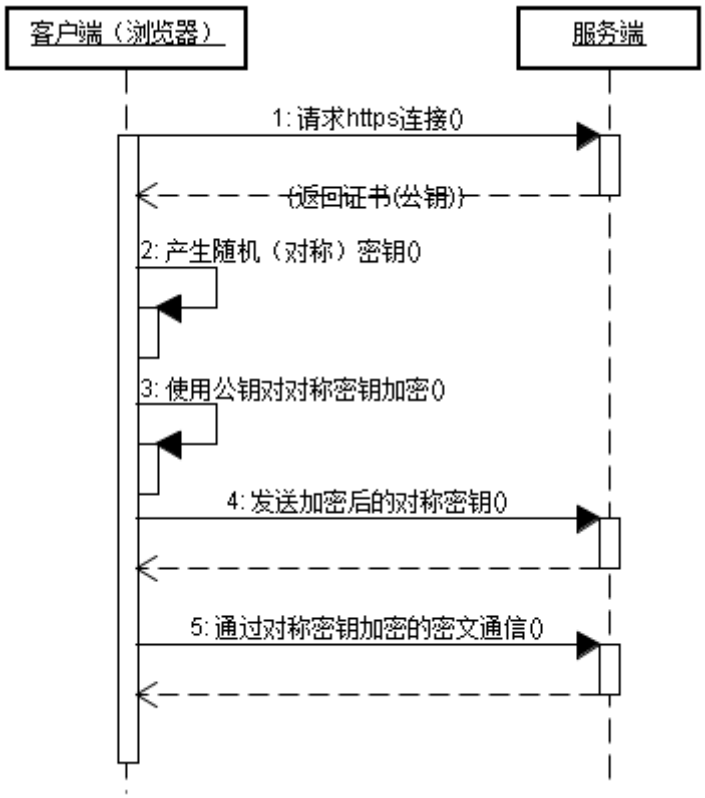
1. 如果服务器没有收到客户端的 ACK，会超时重传 FiN。去向ACK消息最大存活时间(MSL)+ 来向FIN消息的最大存活时间(MSL) = 2msl

2. 如果不等，释放的端口可能会重连刚断开的服务器端口，这样依然存活在网络里的老的 TCP 报文可能与新 TCP 连接报文冲突，造成数据冲突。

参考资料：为什么TCP4次挥手时等待为2MSL？ - 知乎 (zhihu.com)

http和https的区别

https=http+ssl（加密数据，认证服务器）



参考资料：HTTP与HTTPS的区别 - 爱笑的蛙蛙 - 博客园 (cnblogs.com)

说几个HTTP状态码

100 Continue

200 OK

301 Moved Permanently，允许更改 method，不过一般用 GET 重定向

308 Permanent Redirect，不允许更改 method

302 Found，允许更改 method，不过一般用 GET 重定向

303 See Other，使用 GET 重定向

307 Temporary Redirect，不允许更改 method

304 Not Modified

403 Forbidden

404 Not Found

500 Internal Server Error

503 Service Unavailable

参考资料：HTTP状态码(响应码) - 小马奔腾！ - 博客园 (cnblogs.com)

当你用浏览器打开一个链接的时候，计算机做了哪些工作步骤

输入谷歌网址

查找DNS缓存

发起DNS查询

ARP请求

封装 TCP 数据包

浏览器与目标服务器建立 TCP 连接

浏览器发送 HTTP 请求到 web 服务器

服务器处理请求并发回一个响应

服务器发送回一个 HTTP 响应

浏览器显示 HTML 的相关内容

参考资料：当你用浏览器打开一个链接的时候，计算机做了哪些工作*wh*柒八九的博客-*CSDN*博客浏览器链接

如何避免浏览器缓存

请求头 Pragma:no-cache 或 Cache-Control:no-cache。服务端告诉浏览器不要缓存

请求头 Expires:Sat,25Feb201212:22:17GMT。服务端告诉浏览器缓存什么时候过期

请求头 Last-Modified:Sat,25Feb201212:55:04GMT，If-Modified-Since:Sat,25Feb 201212:55:04GMT。服务端返回资源最后修改时间，浏览器请求是询问是否资源最新，是的话服务端返回 304。

请求头Etag：同 Last-Modified。

参考资料：浏览器缓存机制详解 - 李某龙 - 博客园 (cnblogs.com)

如何理解HTTP协议的无状态性

每一个请求都按独立的新请求处理

HTTP有哪几种method

GET：请求已经存在的资源

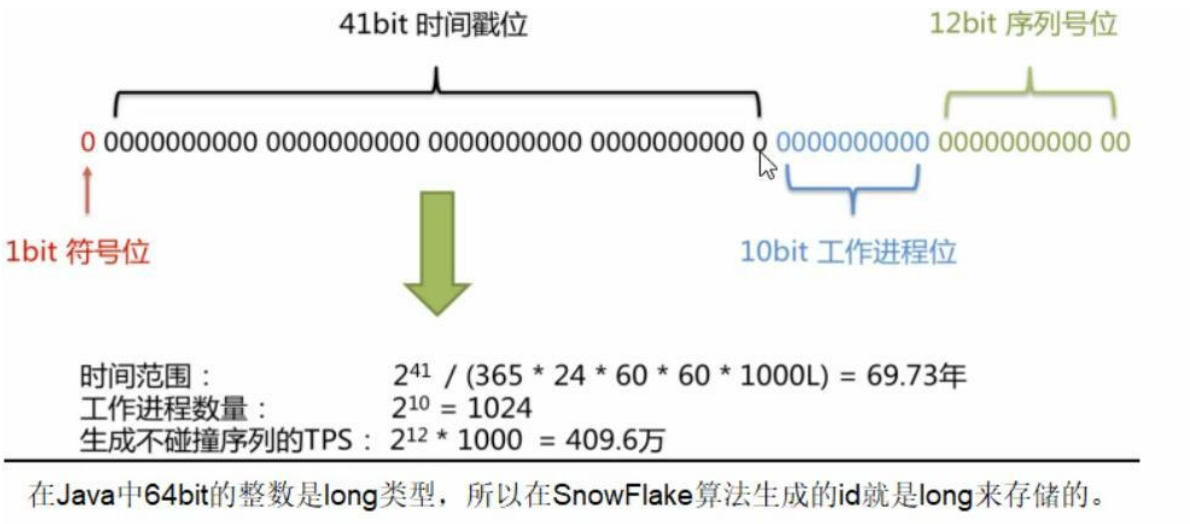
POST：创建新资源或更新资源，不幂等

PUT：创建新资源或更新资源，幂等

DELETE：删除资源

算法

雪花算法



41 位时间戳可以取当前时间戳减去 2015-01-01

10 位机器位 5 位数据中心可以取机器的 hostName % 32，5 位机器 id 取机器的 ip % 32。机器码怎么保证唯一？是否可以用数据库或 redis 做计数器。

12 位序列号位要注意避免偶数偏多情况，可以取 timestamp & 1L

图片来源：分布式ID生成算法-雪花算法 - 彼岸舞 - 博客园 (cnblogs.com)

雪花算法如何解决时钟回拨

1. 直接抛异常
2. 延迟等待，阻塞重新获取时间，如果还是不对，则抛出异常
3. 采用 lastTimestamp，如果 seq 不够，lastTimestamp + 1 √

一致性哈希算法

参考资料：一致性哈希(hash)算法 - 明志健致远 - 博客园 (cnblogs.com)

对称加密、非对称加密算法

时间轮算法

解决hash冲突的方法

再散列法

多重散列

拉链法

建立公共溢出区

paxos算法*

raft算法*

设计

设计模式

单例模式：复用类实例

代理模式：aop 编程

装饰模式：功能增强

简单工厂模式：解耦实现类

模板模式：定义好主流程，变化的地方交给你实现

策略模式：不同扩展点

.....

手写单例模式

懒汉模式

```
public class A {
    private static A instance = new A();

    private A(){}

    public static A getInstance(){
        return instance;
    }
}
```

饿汉模式

```
public class A {
    private static A instance;

    private A(){}
    // 可使用Double-checked_locking优化
    public static synchronized A getInstance(){
        if(instance == null){
           instance = new A();
        }
        return instance;
    }
}
```

静态内部类

```
public class A {
    private A(){}

    public static A getInstance(){
        return AHolder.instance;
    }

    private static class AHolder{
        private static final A instance = new A();
    }
}
```

代理模式和装饰模式的区别

代理模式是控制对象的访问。

装饰模式是给对象增加功能。

面向对象设计的原则

单一职责：就一个类而言，应该仅有一个引起它变化的原因

开放封闭：软件实体（类、模块、函数等）应该是可以扩展的，但是不可修改的

接口隔离：不能强迫用户去依赖那些他们不使用的接口

里氏替换：子类型必须能够替换掉它们的基类型

依赖倒置：高层模块不应该依赖于低层模块，两者都应该依赖于抽象；抽象不应该依赖于细节，细节应该依赖于抽象

如何设计一个良好的API

我的看法：

1. 不要让调用方做逻辑；例如，当 xxx 时这么传，xxx 时那么传
2. 不要信任调用方；例如，认为调用方传的都是合法的
3. 入参精炼；例如，传了 shopId，还要求传 orgId
4. 功能明确；例如，调用方只需要知道你能干嘛，而不需要关注你的其他细节。
5. 兼容更多的场景；

数据库设计的三大范式

第一范式：一个字段只存储一项信息

第二范式：任意一个字段都依赖于同一个主键

第三范式：一个数据库表中不包含已在其它表中已包含的非主键字段

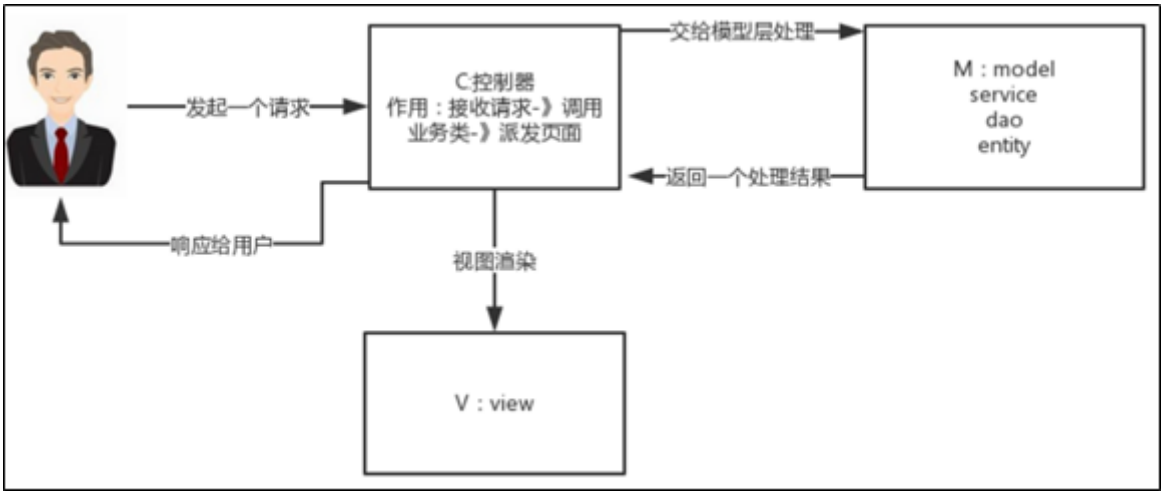
参考资料：mysql 数据库的设计三范式 - cool小伙 - 博客园 (cnblogs.com)

开发、设计中遇到的问题或挑战

最大的挑战就是重构订单中心。首先为什么要重构呢？因为真的很难维护，那是什么导致它很难维护呢：

1. **部分表不遵循第一范式。**存在一个字段存储多种数据的情况。
2. **场景越多，代码里出现了大量穿插的 if 判断，而且经常无法区分哪个场景执行了哪个逻辑**；根据开闭原则，应该对修改关闭，对扩展开放，于是有了我说的代码模型。
3. 代码流程缺少进一步的分层抽象。例如，校验细分为必传、规则、合法性、其他等。
4. 让接口调用方做逻辑。例如，订单页面展示。
5. 太过信任调用方。
6. 入参太不精炼。

什么是MVC



参考资料：[什么是mvc模式 - ChauncyNong - 博客园 (cnblogs.com)]

场景题

实现：抢红包

```
int flag = redisUtils.decr(flagKey + redPacketId);
if(flag < 0){
    // 快速失败
}
redisUtils.lock(resourceKey + redPacketId);
try {
    // 获取红包总个数、当前个数、余额

    // 计算当前抢到余额zzs001

    // 写库：更新余额、剩余数量，以及抢红包记录

} catch (Exception e) {
    redisUtils.incr(flagKey + redPacketId);
    throw e;
} finally{
    redisUtils.unlock(resourceKey + redPacketId);
}
```

实现：扫码登录

参考资料：[面试官：如何实现扫码登录功能？ - 三分恶 - 博客园 (cnblogs.com)]

实现：秒杀*

实现：多少人在看这篇文章*

webSocket?

redis

实现：浏览过这篇文章的还浏览了

实现：附近的人

参考文献：Java中“附近的人”实现方案讨论及代码实现 - larscheng - 博客园 (cnblogs.com)

实现：五亿数据，找出数量top100

实现：并发安全的链表*

实现：30分钟没付款就自动关闭交易

实现：分布式环境下的countDownLatch

其他

长链接转短链接的原理

服务端存长短链接的映射，为了保证短链接唯一且存越多，可以采用高进制的自增序列方案。至于自增序列的具体实现嘛，可以用数据库、redis、雪花等。

tomcat的并发控制参数

```
server:
  #tomcat配置
  tomcat:
    # 达到max-connections后，允许多少连接等待
    accept-count: 100
    # 同一时间能处理的最大连接数（BIO时等于maxPoolSize）
    max-connections: 10000
    # 类似corePoolSize
    min-spare-threads: 10
    # 类似maxPoolSize
    max-threads: 200
```

参考资料：Apache Tomcat 8 Configuration Reference (8.5.78) - The HTTP Connector

结语

感谢阅读。

打赏支持 ~

微信|支付宝

> 本文为原创文章，转载请附上原文出处链接：https://www.cnblogs.com/ZhangZiSheng001/p/16218179.html