基础

数据结构

基本包括

HashMap

数据结构：散列桶数组 + 链表

一般简单的HashMap，大致流程，key值经过计算打散到散列数组对应位置，以链表形式挨个比对完成对应操作

主要了解：**扰动函数、初始化容量、负载因子、扩容方法以及链表和红黑树转换的使用**

扰动函数

散列数组的index，为什么不直接Key的hashCode值取余length，而是多了一次扰动计算

> **String的hashCode方法，为什么选择31作为乘积值?**
>
> 31为奇质数，偶数会导致乘积运算时数据溢出；31 = 2<<5-1,乘积运算可以使用位移提升性能；
>
> hash碰撞概率小（运算过程在int取值范围内），基本稳定。

HashMap，key通过Hash计算索引位置，然后找到对应元素或链表或树，为什么不直接取余(&(size-1))?

key的hashCode，进行一次**扰动计算**，哈希值右移16位（默认初始化大小16长度），与原哈希做异或，这样混合了原哈希值中的高位和地位，增大了**随机性**，进一步**减少了哈希碰撞**，分布的更均匀

初始化容量和负载因子

负载因子默认0.75，就是说当阀值容量占了3/4 时赶紧扩容，减少 Hash 碰撞

扩容元素拆分

扩容方式就是需要把元素拆分到新的数组中。

拆分元素的过程中，原 jdk1.7 中会需要重新计算哈希值，但是到 jdk1.8 中已经进行优化，不在需要重新计算

基本的数据操作功能：存储、删除、获取、遍历

- 存储删除操作可能会遇到哈希碰撞（equals，替换或者插入（树化）），可能会遇到扩容，扩容后，原来因为哈希碰撞存放的链表或者红黑树，都需要进行拆分存放

ArrayList

数据结构：数组

创建一个集合，初始容量不指定时为0，在第一次添加元素时，集合会扩容为10；继续添加至超过上限，会继续扩容为原来的3/2，也就是15

Arrays.asList构建的集合，不能再添加，不能再删除

JVM性能调优

JVM参数配置最佳实践

在一个机器中JVM进程占用总内存(堆，元空间，堆外内存，CodeCache)一般不建议超过总内存的80%

JVM参数	说明	4c8g	8c16g
-Xms	初始堆内存大小	4g	10g
-Xmx	最大堆内存大小	4g	10g
-Xmn	新生代空间大小	2g	5g
-XX:MetaspaceSize	初始元空间大小	384m	512m
-XX:MaxMetaspaceSize	最大元空间大小	384m	512m
-XX:MaxDirectMemorySize	最大堆外内存大小	1g	1g
-XX:ReservedCodeCacheSize	CodeCache(本地方法)大小	256m	256m

不分机器的通用配置

-XX:SurvivorRatio=10，survivor区大小为新生代的1/12

-XX:ParallelGCThreads=${CPU_COUNT}，并发GC线程数（Young GC 在jdk1.8默认使用多线程线性收集器）

-XX:+UseConcMarkSweepGC ，old区使用CMS GC

-XX:CMSMaxAbortablePrecleanTime=5000， CMS GC的回收超时时间，避免GC太久

-XX:CMSInitiatingOccupancyFraction=80 ，Old区达到80%触发CMS GC，如果不设置，JVM会自适应，效果不好

-XX:UserCMSInitiatingOccupancyOnly，必须配置这个，上面这个Fraction配置才会生效

-XX:+CMSScavengeBeforeRemark，执行GMS GC的remark前先Young GC一次，缩短remark时间（会有跨代引用，老年代进行GC Roots追踪时，同样会扫描年轻代）

-XX:+ExplicitGCInvokesConcurrent， 调用System.gc()时触发CMS GC而不是Full GC

-XX:HeapDumpOnOutOfMemoryError，OOM时自动jmap dump内存

-XX:HeapDumpPath=/home/admin/logs/java.hprof，OOM时dump内存的位置

-Xloggc:/home/admin/logs/gc.log，GC log位置

-XX:+PrintGCDetails，打印GC详细信息

-XX:+PrintGCDeteStamps，将GC时间戳改为人类可识别的时间刻

-XX:+UseGCLogFileRotation，开启GC日志文件轮询

-XX:GCLogFileSize=50M，每个GC日志文件最大大小

-XX:NumberOfGCLogFiles=5，保留的GC日志文件个数

多线程顺序执行

- join方法等待线程销毁，阻塞主线程
- CountDownLatch倒数计时器

```
CountDownLatch countDownLatch1 = new CountDownLatch(1);
countDownLatch1.await();
doSomething();
countDownLatch2.countDown();
```

- 单线程线程池，newSingleThreadExecutor()

强引用，软引用，弱引用，虚引用

软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生

程序里面最快的缓存就是直接在内存里面构建缓存。但是如果我们使用强引用构建缓存的话，一旦缓存过大无法回收就会导致内存溢出。这个时候我么可以使用软引用构建缓存。

```
// 软引用
Caffeine.newBuilder().softValues().build();
```

JVM垃圾收集器

Serial（串行）收集器（**新生代采用复制算法，老年代采用标记-整理算法**）

ParNew 收集器（ Serial 收集器的多线程版本）

Parallel Scavenge 收集器（与ParNew几乎一样，关注点吞吐量，高效率利用CPU）**JDK1.8 默认收集器**

CMS（Concurrent Mark Sweep）收集器，最短回收停顿时间为目标的收集器。注重用户体验，“标记-清除”算法(并发收集、低停顿)，参考https://blog.csdn.net/zqz_zqz/article/details/70568819

G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征

ZGC（The Z Garbage Collector）是JDK 11中推出的一款低延迟垃圾回收器(在 ZGC 中出现 Stop The World 的情况会更少，目标10ms)

TCP三次握手四次挥手

三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收是正常的。

断开，多一次是因为服务器再知道断开请求之后还会传完一小部分数据，然后在断

在浏览器中输入url地址 ->> 显示主页的过程

总体来说分为以下几个过程:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

线程安全

jvm内存模型，jvm线程模型

java内存模型

得出线程安全讲的是什么？线程是什么，什么又叫做安全？

https://blog.csdn.net/u014730165/article/details/81981154

ThreadLocal会有什么问题

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。

由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，**发生GC时弱引用Key会被回收，而Value不会回收。**

当线程没有结束，但是ThreadLocal已经被回收，则可能导致线程中存在ThreadLocalMap的键值对，**造成内存泄露。**（`ThreadLocal被回收，ThreadLocal关联的线程共享变量还存在`）

避免泄露两种方法

- 使用完线程共享变量后，显示调用ThreadLocalMap.remove方法清除线程共享变量；
- JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了

MYSQL索引

MYSQL索引的数据结构，B+Tree，哈希

MYSQL引擎，MyISAM，InnoDB

https://www.cnblogs.com/jiawen010/p/11805241.html

InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，聚簇索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。这个特性决定了索引组织表中数据也是索引的一部分；

我们日常工作中，根据实际情况自行添加的索引都是辅助索引，辅助索引就是一个为了需找主键索引的二级索引，现在找到主键索引再通过主键索引找数据；

MyISAM与InnoDB的对比：

1. **是否支持行级锁** : MyISAM 只有表级锁(table-level locking)，而InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁。
2. InnoDB使用的是**聚簇索引**，MyISM使用的是**非聚簇索引**
3. **是否支持事务和崩溃后的安全恢复： MyISAM** 强调的是性能，每次查询具有原子性,其执行速度比InnoDB类型更快，但是不提供事务支持。但是**InnoDB** 提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。
4. **是否支持外键：** MyISAM不支持，而InnoDB支持。
5. **是否支持MVCC** ： 仅 InnoDB 支持。应对高并发事务, MVCC比单纯的加锁更高效;MVCC只在 `READ COMMITTED` 和 `REPEATABLE READ` 两个隔离级别下工作;MVCC可以使用 乐观(optimistic)锁和 悲观(pessimistic)锁来实现;各数据库中MVCC实现并不统一。

Spring Cloud基础组件

> 注册与发现（Eureka,Zookeeper,Consul）
>
> 负载均衡（Ribbon）
>
> Api网关（Gateway, Zuul）
>
> REST调用（Feign）
>
> 容错处理（Hystrix）
>
> 统一管理配置（Config, Apollo,Nacos）
>
> 服务跟踪（Sleuth）

微服务组件——**网关**

一般情况下，**网关**一般都会提供请求转发、安全认证（身份/权限认证）、流量控制、负载均衡、容灾、日志、监控这些功能

网关实现核心，过滤器，路由

常用网关，Kong 基于 Openresty ，Zuul 基于 Java，SpringGateway

基于云平台的产生的概念Service Mesh（服务网格），相较传统部署来说，云平台优点弹性伸缩，相当于云平台的智能网关；

常用Service Mesh，linkerd基于主机部署， istio仅应用与Kubernetes

限流算法：固定窗口，滑动窗口，漏桶，令牌桶

微服务组件——**注册中心**

主要功能，服务注册和服务发现，监控，负载均衡

典型的分布式系统，需要支持高可用

CAP原则指的是在一个分布式系统中，Consistency(一致性)、Availability(可用性)、Partition Tolerance(分区容错性)，不能同时成立

一致性和可用性不能同时满足。在注册中心的发展上面，一直有两个分支：一个就是 CP 系统，追求数据的强一致性。还有一个是 AP 系统，追求高可用与最终一致。我们接下来介绍的服务注册和发现组件中，Eureka满足了其中的AP，Consul和Zookeeper满足了其中的CP

服务注册 Register：

官方解释：当 `Eureka` 客户端向 `Eureka Server` 注册时，它提供自身的**元数据**，比如IP地址、端口，运行状况指示符URL，主页等

服务续约 Renew：

官方解释：`Eureka` **客户会每隔30秒(默认情况下)发送一次心跳来续约。** 通过续约来告知 `Eureka Server` 该 `Eureka` 客户仍然存在，没有出现问题。 正常情况下，如果 `Eureka Server` 在90秒没有收到 `Eureka` 客户的续约，它会将实例从其注册表中删除

获取注册列表信息 Fetch Registries：

官方解释：`Eureka` 客户端从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每30秒钟）更新一次。每次返回注册列表信息可能与 `Eureka` 客户端的缓存信息不同，`Eureka` 客户端自动处理。如果由于某种原因导致注册列表信息不能及时匹配，`Eureka` 客户端则会重新获取整个注册表信息。 `Eureka` 服务器缓存注册列表信息，整个注册表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。`Eureka` 客户端和 `Eureka` 服务器可以使用JSON / XML格式进行通讯。在默认的情况下 `Eureka` 客户端使用压缩 `JSON` 格式来获取注册列表的信息。

服务下线 Cancel：

官方解释：Eureka客户端在程序关闭时向Eureka服务器发送取消请求。 发送请求后，该客户端实例信息将从服务器的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：
`DiscoveryManager.getInstance().shutdownComponent();`

服务剔除 Eviction：

官方解释：在默认的情况下，**当Eureka客户端连续90秒(3个续约周期)没有向Eureka服务器发送服务续约，即心跳，Eureka服务器会将该服务实例从服务注册列表删除**，即服务剔除。

zk扩展 https://blog.csdn.net/dwdyoung/article/details/80726883

微服务组件——**负载均衡**（Ribbon）

> `RestTemplate` **是** `Spring` **提供的一个访问Http服务的客户端类**
>
> 微服务之间的调用是使用的 `RestTemplate`
>
> 上面我们所讲的 `Eureka` 框架中的 **注册**、**续约** 等，底层都是使用的 `RestTemplate`

运行在消费者端，也就是调用方，其工作原理就是 `Consumer` 端获取到了所有的服务列表之后，在其**内部**使用**负载均衡算法**，进行对多个系统的调用

请注意 `Request` 的位置，在 `Nginx` 中请求是先进入负载均衡器，而在 `Ribbon` 中是先在客户端进行负载均衡才进行请求的。所以 `Ribbon` 是一种**集中式**的负载均衡器

负载均衡器Ribbon调度算法

- `RoundRobinRule`：轮询策略。`Ribbon` 默认采用的策略。若经过一轮轮询没有找到可用的 `provider`，其最多轮询 10 轮。若最终还没有找到，则返回 `null`。
- `RandomRule`：随机策略，从所有可用的 `provider` 中随机选择一个。
- `RetryRule`：重试策略。先按照 `RoundRobinRule` 策略获取 `provider`，若获取失败，则在指定的时限内重试。默认的时限为 500 毫秒。

微服务组件——**REST调用**（Open Feign）

理论上有了 `Eureka` ， `RestTemplate` ， `Ribbon` ， 我们就可以愉快地进行服务间的调用；但不方便；故利用映射，注解简化服务间调用

`OpenFeign` 也是运行在消费者端的，使用 `Ribbon` 进行负载均衡，所以 `OpenFeign` 直接内置了 `Ribbon`

微服务组件——**容错处理**（Hystrix之熔断和降级）

> 在分布式环境中，不可避免地会有许多服务依赖项中的某些失败。Hystrix是一个库，可通过添加等待时间容限和容错逻辑来帮助您控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点，停止服务之间的级联故障并提供后备选项来实现此目的，所有这些都可以提高系统的整体弹性。

A --> B --> C C程序有问题，导致大量请求阻塞；过一会之后，B会因为C的阻塞而变得阻塞，同理，A也会。这种现象叫做**服务雪崩**

熔断 就是指的 `Hystrix` 中的 **断路器模式** ，你可以使用简单的 `@HystrixCommand` 注解来标注某个方法

```
@HystrixCommand(
    commandProperties = {@HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds",value = "1200")}
)
public List<Xxx> getXxxx() {
    // ...省略代码逻辑
}
```

降级，是一种比断路模式更友好的一种模式，可指定后背方法进行安抚用户

```
// 指定了后备方法调用
@HystrixCommand(fallbackMethod = "getHystrixNews")
```

Hystrix隔离方式

支持**信号量**隔离和**线程池**隔离

- 信号量隔离适应非网络请求，因为是同步的请求，无法支持超时，只能依靠协议本身
- 线程池隔离，即，每个实例都增加个线程池进行隔离

> 信号量隔离
>
> 每次调用线程，当前请求通过计数信号量进行限制，当信号大于了最大请求数（maxConcurrentRequests）时，进行限制，调用fallback接口快速返回
>
> 最重要的是，**信号量的调用是同步的**，也就是说，每次调用都得阻塞调用方的线程，直到结果返回。这样就导致了无法对访问做超时（只能依靠调用协议超时，无法主动释放）

> 线程池隔离
>
> 通过每次都开启一个单独线程运行。它的隔离是通过线程池，即每个隔离粒度都是个线程池，互相不干扰
>
> 线程池隔离方式，等于多了一层的保护措施，可以通过hytrix直接设置超时，超时后直接返回

微服务组件——**统一管理配置，配置中心**（Spring Cloud Config）

简单来说，`Spring Cloud Config` 就是能将各个 应用/系统/模块 的配置文件存放到 **统一的地方然后进行管理**(Git 或者 SVN)。

你想一下，我们的应用是不是只有启动的时候才会进行配置文件的加载，那么我们的 `Spring Cloud Config` 就暴露出一个接口给启动应用来获取它所想要的配置文件，应用获取到配置文件然后再进行它的初始化工作。

可以*使用 Spring Cloud Bus （需要借助消息Kafka）来自动刷新多个端*

ElasticsSearch vs MongoDB

https://www.sohu.com/a/283594658_505800

es和mongoDB分片及高可用对比

https://www.cnblogs.com/lazio10000/p/8111719.html

java线程通信方式

- **“共享内存”式的通信**
- **Object类的 wait() 和 notify() 方法**，Condition的await()和signal()方法
- **管道通信**就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

es倒排索引

分词后，作为键值，倒排列表为ids主键集合（1.2.3.5）

写操作MongoDB比传统数据库快的根本原因是Mongo使用的内存映射技术 - 写入数据时候只要在内存里完成就可以返回给应用程序，这样并发量自然就很高。而保存到硬体的操作则在后台异步完成。注意MongoDB在2.4就已经是默认安全写了（具体实现在驱动程序里），所以楼上有同学的回答说是”默认不安全“应该是基于2.2或之前版本的。

读操作MongoDB快的原因是： 1）MongoDB的设计要求你常用的数据（working set)可以在内存里装下。这样大部分操作只需要读内存，自然很快。 2）文档性模式设计一般会是的你所需要的数据都相对集中在一起（内存或硬盘），大家知道硬盘读写耗时最多是随机读写所产生的磁头定位时间，数据集中在一起则减少了关系性数据库需要从各个地方去把数据找过来（然后Join）所耗费的随机读时间

另外一个就是如@王子亭所提到的Mongo是分布式集群所以可以平行扩展。目前一般的百万次并发量都是通过几十上百个节点的集群同时实现。这一点MySQL基本无法做到（或者要花很大定制的代价）

mongodb索引 采用的wiredTiger 引擎，是按照b-tree(2-3树)的形式来组织的，进行了扩展，叶子节点存储了key 和 数据

B-Tree是为磁盘或其它辅助存储设备而设计的一种数据结构，目的是为了在查找数据的过程中减少磁盘I/O的次数

在整个B-Tree中，从上往下依次为Root结点、内部结点和叶子结点，每个结点就是一个Page，数据以Page为单位在内存和磁盘间进行调度，每个Page的大小决定了相应结点的分支数量，每条索引记录会包含一个数据指针，指向一条数据记录所在文件的偏移量。

为什么 MongoDB （索引）使用B-树而 Mysql 使用 B+树

B-树是一类树，包括B-树、B+树、B*树等，是一棵自平衡的搜索树，它类似普通的平衡二叉树，不同的一点是B-树允许每个节点有更多的子节点。B-树是专门为外部存储器设计的，如磁盘，它对于读取和写入大块数据有良好的性能，所以一般被用在文件系统及数据库中

平衡二叉树有很多，如 AVL 树，红黑树等。这些树在一般情况下查询性能非常好，但当数据非常大的时候它们就无能为力了。原因当数据量非常大时，内存不够用，大部分数据只能存放在磁盘上，只有需要的数据才加载到内存中。平衡二叉树高度相对较大log n，逻辑上很近的节点可能会比较远，无法很好利用磁盘预读（局部性原理），故在数据库和文件系统不使用平衡二叉树

B+树是B-树的变种，它与B-树的不同之处在于：

- 在B+树中，key 的副本存储在内部节点，真正的 key 和 data 存储在叶子节点上
- n 个 key 值的节点指针域为 n 而不是 n+1

内节点并不存储 data，所以一般B+树的叶节点和内节点大小不同，而B-树的每个节点大小一般是相同的，为一页。

为了增加 区间访问性，一般会对B+树做一些优化

B树的概念

https://blog.csdn.net/v_JULY_v/article/details/6530142

B-树和B+树的区别

1.B+树内节点不存储数据，所有 data 存储在叶节点导致查询时间复杂度固定为 log n。而B-树查询时间复杂度不固定，与 key 在树中的位置有关，最好为O(1)。

2.B+树叶节点两两相连可大大增加区间访问性，可使用在范围查询等，而B-树每个节点 key 和 data 在一起，则无法区间查找

3.B+树更适合外部存储。由于内节点无 data 域，每个节点能索引的范围更大更精确

为什么 MongoDB 索引选择B-树，而 Mysql 索引选择B+树

- MongoDB 是一种 nosql，也存储在磁盘上，被设计用在 数据模型简单，性能要求高的场合。MongoDB 是聚合型数据库，而 B-树恰好 key 和 data 域聚合在一起
- Mysql 是一种关系型数据库，区间访问是常见的一种情况，而 B-树并不支持区间访问（可参见上图），而B+树由于数据全部存储在叶子节点，并且通过指针串在一起，这样就很容易的进行区间遍历甚至全部遍历

缓存是否可以使用堆外内存 https://www.jianshu.com/p/17e72bb01bf1

OHC，全称为 **off-heap-cache**，即堆外缓存，是一款基于Java 的 key-value 堆外缓存框架

与堆内空间不同，堆外空间不影响GC，由应用程序自身负责分配与释放内存

两种分配堆外内存的方法，Unsafe和NIO；

caffinitas，OHC是将Java对象序列化后存储在堆外，因此用户需要实现 org.caffinitas.ohc.CacheSerializer 类，OHC会运用其实现类来序列化和反序列化对象。

https://www.cnblogs.com/liang1101/p/13499781.html

https://my.oschina.net/u/4384701/blog/3386255

MySQL，binlog日志有哪几种，如何利用binlog同步es或者从库

MySQL binlog日志有三种格式，分别为Statement,MiXED,以及ROW

- **Statement**：每一条会修改数据的sql都会记录在binlog中
- **Row:**不记录sql语句上下文相关信息，仅保存哪条记录被修改
- **Mixedlevel:** 是以上两种level的混合使用，一般的语句修改使用statment格式保存binlog，statement无法完成主从复制的操作，则采用row格式保存binlog

消息中间件，有没有遇到丢失的情况 https://www.jianshu.com/p/4491cba335d1

- 生产者丢了数据
  - RabbitMQ 提供的事务功能，生产者发送，服务端未成功接收，生产者回滚，强一致性，一般不采用
  - RabbitMQ开启confirm模式，前者同步的，这个是异步的，发送消息后，等待服务器接收后的ack异步回调，一般用confirm模式
  - **[Kafka]**producer端设置acks=all，要求每条数据，必须是**写入所有 replica 之后，才能认为是写成功了**
- 消息中间件丢了数据
  - 开启持久化，RabbitMQ开启持久化
  - **[Kafka]**topic设置 `replication.factor` 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本； **&** Kafka 服务端设置 `min.insync.replicas` 参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系（场景：leader 挂掉，follow切leader，可能会丢数据；）
- 消费端弄丢了数据
  - 关闭RabbitMQ的自动ACK，手动消费处理完后，手动调用ACK一次；防止未消费结果进程挂了，却自动ACK。注意保持幂等
  - **[Kafka]**关闭自动提交offset，在处理完之后自己手动提交 offset，就可以保证数据不会丢。注意保持幂等

线程池的核心参数 https://blog.csdn.net/lveex/article/details/78261173

- corePollSize：核心线程数。在创建了线程池后，线程中没有任何线程，等到有任务到来时才创建线程去执行任务。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中。
- maximumPoolSize：最大线程数。表明线程中最多能够创建的线程数量。
- keepAliveTime：空闲的线程保留的时间，TimeUnit：空闲线程的保留时间单位
- BlockingQueue：阻塞队列，存储等待执行的任务。参数有ArrayBlockingQueue（有界队列）、LinkedBlockingQueue（无界队列，可产生OOM）、SynchronousQueue可选。
- ThreadFactory：线程工厂，用来创建线程，新的线程都是由ThreadFactory创建的，系统默认使用的是Executors.defaultThreadFactory创建的，用它创建出来的线程的优先级、组等都是一样的，并且他都不是守护线程。我们也可以使用自定义的线程创建工厂，并对相关的值进行修改
- RejectedExecutionHandler：队列已满，而且任务量大于最大线程的异常处理策略
    - ThreadPoolExecutor.AbortPolicy：丢弃任务并抛出RejectedExecutionException异常
    - ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常
    - ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
    - ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

Synchonize vs ThreadLocal区别

都是解决线程安全问题，Sync解决共享变量并发访问的线程安全的方式是：线性挨个执行；
ThreadLocal是利用线程间的隔离，独立享有各自的内存达到线程安全

Redis部署

HashMap为什么线程不安全

- 在JDK1.7中，当并发执行扩容操作时会造成环形链和数据丢失的情况
- 在JDK1.8中，在并发执行put操作时会发生数据覆盖的情况

HashSet数据结构

ConcurrentHashMap为什么去掉Segment

- 分段锁的优势在于保证在操作不同段 map 的时候可以并发执行，操作同段 map 的时候，进行锁的竞争和等待。这相对于直接对整个map同步synchronized是有优势的。
- 缺点在于**分成很多段时会比较浪费内存空间**(不连续，碎片化); 操作map时竞争同一个分段锁的概率非常小时，分段锁反而会造成更新等操作的长时间等待; 当某个段很大时，**分段锁的性能**会下降

Hystrixs 隔离实现

信号量，线程池

因为信号量同步，无法支持超时，一般选用线程池，多了一层，需要配置好

线程池最佳实践

线程池资源优化演进

在生产环境中部署一个短路器，一开始需要将一些关键配置设置的大一些，比如timeout超时时长，线程池大小，或信号量容量，然后逐渐优化这些配置，直到在一个生产系统中运作良好。

（1）一开始先不要设置timeout超时时长，默认就是1000ms，也就是1s
（2）一开始也不要设置线程池大小，默认就是10
（3）直接部署hystrix到生产环境，如果运行的很良好，那么就让它这样运行好了
（4）让hystrix应用，24小时运行在生产环境中
（5）依赖标准的监控和报警机制来捕获到系统的异常运行情况
（6）在24小时之后，看一下调用延迟的占比，以及流量，来计算出让短路器生效的最小的配置数字
（7）直接对hystrix配置进行热修改，然后继续在hystrix dashboard上监控
（8）看看修改配置后的系统表现有没有改善

HystrixThreadPool线程数量设定计算公式

每秒的高峰访问次数 * 99%的访问延时 + 缓冲数量

假设一个依赖的服务的高峰访问次数30QPS，99%的请求延时实践在200ms，那么我们可以设定的线程数量=30*0.2+4=10个线程

HystrixCommand线程超时timeout设定

结合线程池数量配置，设置能够完成执行高峰时QPS任务的timeout数值。

> 默认线程超时为1s。假设一个依赖服务的高峰访问时，99.5%的请求延时在250ms，假设再允许服务重试一次消耗50ms，那么这个依赖服务的延时设置为300ms就可以了，这样一个线程最多延时300ms，也就是这个线程每秒能执行三次依赖服务，线程池中10个线程，整好能处理30QPS。如果把延时设置为400ms，那么一个线程每秒最多执行2次请求，10个线程在1s内最多执行20个请求，剩余的10个请求就会被线程池阻塞住，再来新的请求也继续会被阻塞住，如果等待队列已满，那么新来的请求就会被直接Reject。所以400ms的超时设置就不能满足30QPS的请求次数要求。

生产环境中动态分配线程池资源

线程池动态配置选项

//允许线程池自动从coreSize扩容到maximumSize

.withAllowMaximumSizeToDivergeFromCoreSize(true)

//default 10，默认线程池大小

.withCoreSize(10)

//最大线程数

.withMaximumSize(30)

//空闲线程存活时间1分钟，超时后线程池数量自动恢复到coreSize大小

.withKeepAliveTimeMinutes(1)

深度不够

谦逊一点就好，无关其他，各方面的事情不用也不可能样样精通，尽量表现出所知所学即可，也表现出谦逊好学。