

微服务

1.什么是微服务？

2.kafka优点和缺点？ 和传统的MQ对比有什么区别？

3.为什么 Kafka 吞吐量高

4.不同微服务独立数据库，如何保障微服务架构下的数据一致性

Redis

1.Redis数据结构

String hash list set sortSet

2.Redis扩容机制与ConcurrentHashMap的扩容策略比较

首先Redis的字典采用的是一种“**单线程渐进式rehash**”，这里的单线程是指只有一个线程在扩容，而在扩容的同时其他的线程可以并发的进行读写。Redis系统后台会定时给予扩容的那个线程足够的运行时间，这样不会导致它饿死。

1.Redis字典扩容机制

1.1 哈希表被扩展、收缩的条件：

1. Redis服务器目前没有在执行BGSAVE命令或BGREWRITEAOF（异步执行一个AOF（AppendOnly File）文件重写操作）命令，并且哈希表的负载因子大于等于1。
2. Redis服务器目前在执行BGSAVE命令或BGREWRITEAOF命令，并且哈希表的负载因子大于等于5。
负载因子=哈希表已保存节点数量 / 哈希表大小 $load_factor = ht[0].used / ht[0].size$
3. 当哈希表的负载因子小于0.1时，对哈希表执行收缩操作。

1.2 rehash的操作步骤

1. 为字典的ht[1]哈希表分配空间。
如果执行的是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used*2的2的n次幂
如果执行的是收缩操作，那么ht[1]的大小为第一个大于等于ht[0].used的2的n次幂
2. 单线程A将ht[0]中的数据copy到ht[1]中，在转移的过程中，重新计算键的哈希值和索引值，然后将键值对放置到ht[1]的指定位置。
转移过程中有其他线程操作：
 - 进行读操作：**会先去ht[0]中找，找不到再去ht[1]中找。**
 - 进行写操作：**直接写在ht[1]中。**
 - 进行删除操作：与读类似。当然这过程中会设计到一系列的锁来保证同步性。
3. 当ht[0]的所有键值对都迁移到了ht[1]之后（ht[0]变为空表），将ht[0]释放，然后将ht[1]设置成ht[0]，最后为ht[1]分配一个空白哈希表。

2.ConcurrentHashMap的扩容策略

ConcurrentHashMap采用的扩容策略为：“多线程协同式rehash”。这里的多线程指的是，有多个线程并发的把数据从旧的容器搬运到新的容器中。

- 扩容时大致过程如下：

线程A在扩容把数据从oldTable搬到newTable，这时其他线程

- 进行get操作：这个线程知道数据存放在oldTable或是newTable中，直接取即可。
- 进行写操作：如果要写的桶位，已经被线程A搬运到了newTable。那么这个线程知道正在扩容，它也一起帮着扩容，扩容完成后才进行put操作。
- 进行删除操作：与写一致。

3.两者对比：

1. 扩容所花费的时间对比：

一个单线程渐进扩容，一个多线程协同扩容。在平均的情况下，是ConcurrentHashMap快。这也意味着，扩容时所需要花费的空间能够更快的进行释放。

2. 读操作，两者的性能相差不多。

3. 写操作，Redis的字典更快些，因为它不像ConcurrentHashMap那样去帮着扩容(当要写的桶位已经搬到了newTable时)，等扩容完才能进行操作。

4. 删除操作，与写一样。

所以是选择单线程渐进式扩容还是选择多线程协同式扩容，这个就具体问题具体分析了：

- 如果内存资源吃紧，希望能够进行快速的扩容方便释放扩容时需要的辅助空间，那么选择后者。
- 如果对于写和删除操作要求迅速，那么可以选择前者。

3.Redis高性能的原因

1. Redis是把数据放在内存中的（内存中数据库存取速度最快），类似HashMap对象。
2. key-value (json，字符串)，数据存储结构简单。
3. Redis为单线程模式，不存在资源竞争问题（内存中的数据，不需要多线程）
4. 多路复用：单个线程来处理很多个请求，按先后顺序放到队列中，先到的请求先处理。
5. resp协议：最简单的指令

set age 5 --> 将此行命令分解成如下指令：

*3 --> 表示有三组

\$3 --> 表示第一组的长度

set --> 代表具体的指令

\$3 --> 表示第二组的长度

age --> 具体指令

\$1 --> 第三组长度

5 --> 具体值或指令

4.Redis持久化

1.1 触发机制

RDB持久化触发机制分为：手动触发和自动触发 手动触发

save命令：会阻塞当前服务器，直到RDB完成为止，如果数据量大的话会造成长时间的阻塞，线上环境一般禁止使用 **bgsave命令**：就是background save，执行bgsave命令时Redis主进程会fork一个子进程来完成RDB的过程，完成后自动结束（操作系统的多进程Copy On Write机制，简称COW）。所以Redis主进程阻塞时间只有fork阶段的那一下。相对于save，阻塞时间很短。

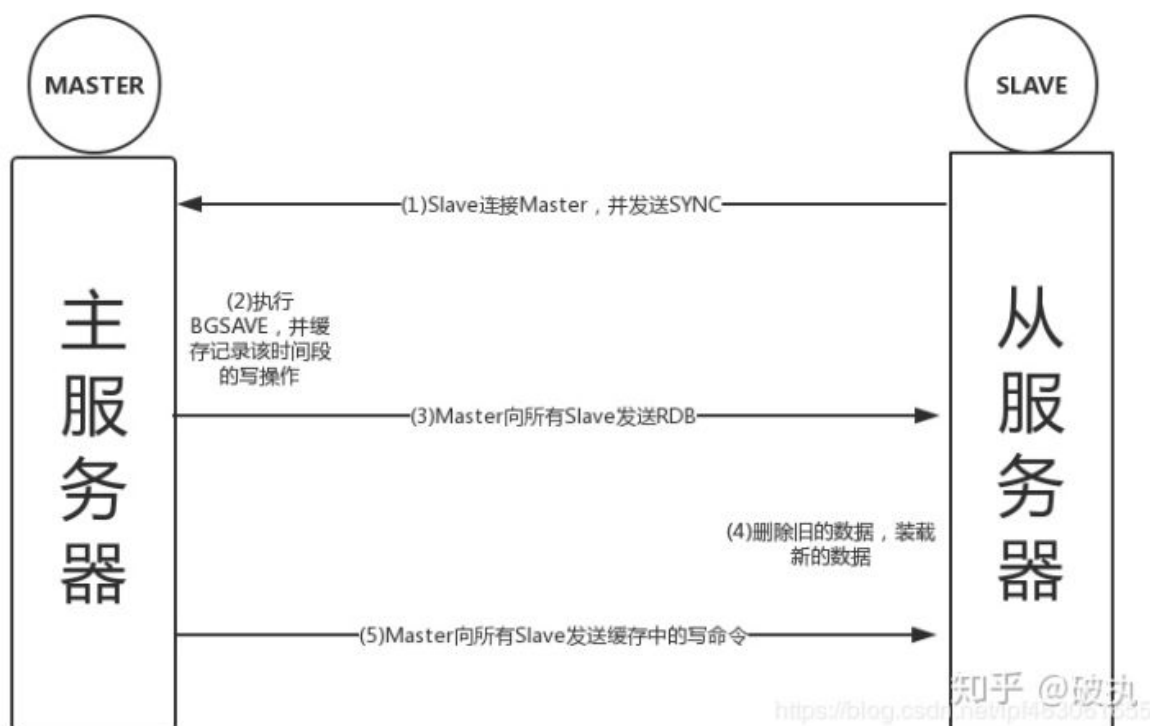
自动触发

场景一：配置redis.conf，触发规则，自动执行

```
# 当在规定的时间内，Redis发生了写操作的个数满足条件，会触发发生BGSAVE命令。
# save <seconds> <changes>
# 当用户设置了多个save的选项配置，只要其中任一条满足，Redis都会触发一次BGSAVE操作
save 900 1
save 300 10
save 60 10000
# 以上配置的含义：900秒之内至少一次写操作、300秒之内至少发生10次写操作、
# 60秒之内发生至少10000次写操作，只要满足任一条件，均会触发bgsave
```

场景二：执行shutdown命令关闭服务器时，如果没有开启AOF持久化功能，那么会自动执行一次bgsave

场景三：主从同步（slave和master建立同步机制）



1.2 RDB执行流程

Redis 使用操作系统的多进程 cow(Copy On Write) 机制来实现RDB快照持久化

1. 执行bgsave命令的时候，Redis主进程会检查是否有子进程在执行RDB/AOF持久化任务，如果有的话，直接返回
2. Redis主进程会fork一个子进程来执行RDB操作，fork操作会对主进程造成阻塞（影响Redis的读写），fork操作完成后会发消息给主进程，从而不再阻塞主进程。（阻塞仅指主进程fork子进程的过程，后续子进程执行操作时不会阻塞）
3. RDB子进程会根据Redis主进程的内存生成临时的快照文件，持久化完成后会使用临时快照文件替换掉原来的RDB文件。（该过程中主进程的读写不受影响，但Redis的写操作不会同步到主进程的

主内存中，而是会写到一个临时的内存区域作为一个副本)

- 子进程完成RDB持久化后会发消息给主进程，通知RDB持久化完成（将上阶段内存副本中的增量写数据同步到主内存）

1.3 RDB的优缺点

优点

- RDB文件小，非常适合定时备份，用于灾难恢复
- Redis加载RDB文件的速度比AOF快很多，因为RDB文件中直接存储的时内存数据，而AOF文件中存储的是一条条命令，需要重演命令。

缺点

- RDB无法做到实时持久化，若在两次bgsave间宕机，则会丢失区间（分钟级）的增量数据，不适用于实时性要求较高的场景
- RDB的cow机制中，fork子进程属于重量级操作，并且会阻塞redis主进程
- 存在老版本的Redis不兼容新版本RDB格式文件的问题

2、AOF (append only file) 日志

AOF日志是持续增量的备份，是基于写命令存储的可读的文本文件。AOF日志会在持续运行中持续增大，由于Redis重启过程需要优先加载AOF日志进行指令重放以恢复数据，恢复时间会无比漫长。所以需要定期进行AOF重写，对AOF日志进行瘦身。目前AOF是Redis持久化的主流方式。

2.1 开启方式

AOF默认是关闭的，通过redis.conf配置文件进行开启

```
## 此选项为aof功能的开关，默认为“no”，可以通过“yes”来开启aof功能
## 只有在“yes”下，aof重写/文件同步等特性才会生效
appendonly yes

## 指定aof文件名称
appendfilename appendonly.aof

## 指定aof操作中文件同步策略，有三个合法值：always everysec no,默认为everysec
appendfsync everysec
## 在aof-rewrite期间，appendfsync是否暂缓文件同步，“no”表示“不暂缓”，“yes”表示“暂缓”，默认为“no”
no-appendfsync-on-rewrite no

## aof文件rewrite触发的最小文件尺寸(mb,gb)，只有大于此aof文件大于此尺寸是才会触发rewrite，默认“64mb”，建议“512mb”
auto-aof-rewrite-min-size 64mb

## 相对于“上一次”rewrite，本次rewrite触发时aof文件应该增长的百分比
## 每一次rewrite之后，redis都会记录下此时“新aof”文件的大小(例如A)
## aof文件增长到A*(1 + p)之后，触发下一次rewrite，每一次aof记录的添加，都会检测当前aof文件的尺寸。
auto-aof-rewrite-percentage 100
```

AOF是文件操作，对于变更操作比较密集的server，那么将造成磁盘IO的负荷加重。此外linux对文件操作采取了“延迟写入”手段，即并非每次write操作都会触发实际磁盘操作，而是进入了buffer中，当buffer数据达到阈值时触发实际写入(也有其他时机)，这是linux对文件系统的优化。

Linux 的 `glibc` 提供了 `fsync(int fd)` 函数可以将指定文件的内容强制从内核缓存刷到磁盘。只要 Redis 进程实时调用 `fsync` 函数就可以保证 aof 日志不丢失。但是 `fsync` 是一个磁盘 IO 操作，它很慢！如果 Redis 执行一条指令就要 `fsync` 一次，那么 Redis 高性能的地位就不保了。

因此在上述配置文件中，可观察到 Redis 中提供了 3 种 AOF 记录同步选项：

- `always`：每一条 AOF 记录都立即同步到文件，性能很低，但较为安全。
- `everysec`：每秒同步一次，性能和安全性都比较中庸的方式，也是 Redis 推荐的方式。如果遇到物理服务器故障，可能导致最多 1 秒的 AOF 记录丢失。
- `no`：Redis 永不直接调用文件同步，而是让操作系统来决定何时同步磁盘。性能较好，但很不安全。

2.2 重写 (rewrite) 机制

AOF 日志会在持续运行中持续增大，需要定期进行 AOF 重写，对 AOF 日志进行瘦身。

AOF Rewrite 虽然是“压缩”AOF 文件的过程，但并非采用“基于原 AOF 文件”来重写或压缩，而是采取了类似 RDB 快照的方式：基于 Copy On Write，全量遍历内存中数据，然后逐个序列到 AOF 文件中。因此 AOF rewrite 能够正确反应当前内存数据的状态。

AOF 重写 (`bgrewriteaof`) 和 RDB 快照写入 (`bgsave`) 过程类似，二者都消耗磁盘 IO。Redis 采取了“schedule”策略：无论是“人工干预”还是系统触发，快照和重写需要逐个被执行。

重写过程中，对于新的变更操作将仍然被写入到原 AOF 文件中，同时这些新的变更操作也会被 Redis 收集起来。当内存中的数据被全部写入到新的 AOF 文件之后，收集的新的变更操作也将被一并追加到新的 AOF 文件中。然后将新 AOF 文件重命名为 `appendonly.aof`，使用新 AOF 文件替换老文件，此后所有的操作都将被写入新的 AOF 文件。

2.3 触发机制

和 RDB 类似，AOF 触发机制也分为：**手动触发**和**自动触发**

手动触发 直接调用 `bgrewriteaof` 命令

```
redis-cli -h ip -p port bgrewriteaof
```

自动触发

根据 `auto-aof-rewrite-min-size` 和 `auto-aof-rewrite-percentage` 参数确定自动触发时机

`auto-aof-rewrite-min-size`: 表示运行 AOF 重写时文件最小体积，默认为 64MB（我们线上是 512MB）。

`auto-aof-rewrite-percentage`: 代表当前 AOF 文件空间 (`aof_current_size`) 和上一次重写后 AOF 文件空间 (`aof_base_size`) 的值

自动触发时机：

```
(aof_current_size > auto-aof-rewrite-min-size) && (aof_current_size -  
aof_base_size) / aof_base_size >= auto-aof-rewrite-percentage
```

其中 `aof_current_size` 和 `aof_base_size` 可以在 `info Persistence` 统计信息中查看。

2.4 AOF 的优缺点

优点 AOF 只是追加写日志文件，对服务器性能影响较小，速度比 RDB 要快，消耗的内存较少

缺点

- AOF方式生成的日志文件太大，需要不断AOF重写，进行瘦身。
- 即使经过AOF重写瘦身，由于文件是文本文件，文件体积较大（相比于RDB的二进制文件）。
- AOF重演命令式的恢复数据，速度显然比RDB要慢。

3、Redis 4.0 混合持久化

- 仅使用RDB快照方式恢复数据，由于快照时间粒度较大，时回丢失大量数据。
- 仅使用AOF重放方式恢复数据，日志性能相对 rdb 来说要慢。在 Redis 实例很大的情况下，启动需要花费很长的时间。

Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——**混合持久化**。将 rdb 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是自持久化开始到持久化结束的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小。相当于：

- 大量数据使用粗粒度（时间上）的rdb快照方式，性能高，恢复时间快。
- 增量数据使用细粒度（时间上）的AOF日志方式，尽量保证数据的不丢失。

在 Redis 重启的时候，可以先加载 rdb 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升。

混合持久化是最佳方式吗？

不一定。

首先，混合持久化是Redis 4.0才引入的特性，现在很多 公司可能都还在使用3.x版本。使用不了这一特性。

另外，可以使用下面这种方式。Master使用AOF，Slave使用RDB快照，master需要首先确保数据完整性，它作为数据备份的第一选择；slave提供只读服务或仅作为备机，它的主要目的就是快速响应客户端read请求或灾切换。

至于具体使用哪种持久化方式，就看大家根据场景选择。没有最好，只有最合适。

5.redis数据过期策略

• 定期删除

指的是 redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除

假如在redis 里插入10w个key，并且都设置了过期时间，如果每次都检查所有key，那cpu基本上都消耗在过期key的检查上了，redis对外的性能也会大大降低

• 惰性删除

惰性删除，就是在获取某个 key 的时候，redis 会检查一下，如果这个 key 设置了过期时间，并且已经过期了，那么就直接删除，返回空。

网络协议和网络编程

1.NIO的好处，Netty线程模型，什么是零拷贝

框架使用

1.Spring中的bean的单例和多例

在Spring中，bean可以被定义为两种模式：prototype（多例）和singleton（单例）

- singleton（单例）：只有一个共享的实例存在，所有对这个bean的请求都会返回这个唯一的实例。
- prototype（多例）：对这个bean的每次请求都会创建一个新的bean实例，类似于new。

Spring bean 默认是单例模式。

单例模式成员变量有线程安全问题，因为成员变量保存在堆中，所有线程都能访问。

数据库

1.数据库的事务，事物的特性，事务的隔离级别分别解决了哪些问题，理解什么是脏读，幻读！事务的实现原理通过什么要保证的事务的特性？

- **数据库事务：**

数据库事务(transaction)，是指作为单个逻辑工作单元执行的一系列操作（对数据库的增删改查），要么完全执行，要么完全不执行。

- **事物的特性：**

- 原子性

原子性是指事务包含的全部数据库操作，要么全部成功，要么全部失败回滚。不存在中间状态。

- 一致性

一致性是指事务必须使数据库从一个一致性状态转换到另一个一致性状态。

- 隔离性

隔离性是指多个用户并发访问数据库时，数据库为每一个请求开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

- 持久性

持久性是指一个事务一旦提交了，那么对数据库数据的操作是永久性的，即便是数据库遇到故障，也不会丢失提交事务的操作。

- **事务的隔离级别分别解决了哪些问题**

1. 什么是左连接，什么是右连接，什么是全连接，什么是内连接？
2. 数据库的索引有什么作用？用什么来实现的？好处坏处是啥？
3. 索引的种类，原理，索引存了哪些内容，什么时候索引会失效？唯一索引和主键索引的区别！单列和联合索引，最左匹配原则，什么时候该用联合索引？
4. 怎么看这个表是否加了索引？
5. B树和B+树有什么区别？为什么索引不用B树？那B+树的叶子结点上存了哪些信息？
6. 数据库的锁？乐观锁悲观锁，共享锁和排它锁。
7. MySql中主要使用的引擎，它们主要的区别是啥。
8. 数据库范式
9. 数据库五大约束？

10. 数据库连接池：工作原理，参数，种类，会出现的问题
11. 数据库的读写分离，数据切分（数据库分库分表，水平切分垂直切分啊）
12. 数据库的主从：实现原理，mysql主从复制的方式，如何配置主从同步，主从同步会出现的问题

Spring:

1. 这一块问的比较少其实，但是IOC和AOP问的很多
2. SpringBoot的启动和运行原
3. IOC和AOP原理？
4. Spring生成代理对象
5. BeanFactory 和 FactoryBean 的区别
6. Spring各个注解的作用？注解的原理？
7. 哪些bean会被扫描？Bean的生命周期！！
8. Spring 实例化 Bean 的过程。
9. Spring 直接注入和直接new一个对象有什么不同？

10.Spring事务管理：事务原理？事务管理接口？实现方式！

1. Spring解决对象相互依赖

Docker学习

1.docker概念：

容器就是将软件打包成标准化单元，以用于开发、交付和部署。

- 容器镜像是轻量的、可执行的独立软件包，包含软件运行所需的所有内容：代码、运行时环境、系统工具、系统库和设置。
- 容器化软件适用于基于Linux和Windows的应用，在任何环境中都能够始终如一地运行。
- 容器赋予了软件独立性，使其免受外在环境差异（例如，开发和预演环境的差异）的影响，从而有助于减少团队间在相同基础设施上运行不同软件时的冲突。

容器虚拟化的是操作系统而不是硬件，容器之间是共享同一套操作系统资源的。虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统。因此容器的隔离级别会稍低一些。

2.为什么要用 Docker？

- Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现“这段代码在我机器上没问题啊”这类问题；——一致的运行环境
- 可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。——更快速的启动时间
- 避免公用的服务器，资源会容易受到其他用户的影响。——隔离性
- 善于处理集中爆发的服务器使用压力；——弹性伸缩，快速扩展
- 可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。——迁移方便
- 使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。——持续交付和部署

3.容器与虚拟机总结

- 容器是一个应用层抽象，用于将代码和依赖资源打包在一起。多个容器可以在同一台机器上运行，共享操作系统内核，但各自作为独立的进程在用户空间中运行。与虚拟机相比，容器占用的空间较少（容器镜像大小通常只有几十兆），瞬间就能完成启动。
- 虚拟机 (VM) 是一个物理硬件层抽象，用于将一台服务器变成多台服务器。管理程序允许多个 VM 在一台机器上运行。每个 VM 都包含一整套操作系统、一个或多个应用、必要的二进制文件和库资源，因此占用大量空间。而且 VM 启动也十分缓慢。

4.Docker基本概念

1. 镜像 (Image) ： 一个特殊的文件系统

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

2. 容器 (Container) ： 镜像运行时的实体

3. 仓库 (Repository) ： 集中存放镜像文件的地方

spring

1.Spring的IOC容器初始化流程

- IOC 控制反转

IOC是存放bean（new OrderServiceImpl()）的容器，spring底层实现数据结构为HashMap：

```
Map<String, Object> map = new HashMap<>;
```

bean存放到IOC容器需要四步：

- 创建bean new OrderServiceImpl()
- bean属性赋值
- bean初始化 -----> IOC MAP
- 回收创建过程的内存，关闭BeanFactory

```
map.put("orderServiceImpl",new OrderServiceImpl());

Controller{

    @Autowired

    OrderServiceImpl orderServiceImpl; //BeanFactory.getBean()

    {map.get("orderServiceImpl");}

}
```

springboot学习

1.springboot:

旨在简化创建产品级的 Spring 应用和服务，简化了配置文件，使用嵌入式web服务器，含有诸多开箱即用微服务功能，可以和spring cloud联合部署。

2.Springboot启动机制

- Springboot集成了SpringMVC，内置了Tomcat。启动Tomcat，集成SpringMVC的流程：通过注解@SpringBootApplication->@EnableAutoConfiguration -> @Import 加载spring-boot-autoconfigure中META-INF/spring.factories中所有的配置类，其中EmbeddedWebServerFactoryCustomizerAutoConfiguration、ServletWebServerFactoryAutoConfiguration内置集成和启动Tomcat，配置类WebMvcAutoConfiguration使用EnableWebMvc集成SpringMVC的功能。

springcloud学习

1.SpringCloud:

微服务工具包，为开发者提供了在分布式系统的配置管理、服务发现、断路器、智能路由、微代理、控制总线等开发工具包。

2.SpringCloud都有哪些组件

- **依赖其它组件并为其提供服务**

Ribbon，客户端负载均衡，特性有区域亲和、重试机制。

Hystrix，熔断器，实现断路器模式，通过控制服务的节点,从而对延迟和故障提供更强大的容错能力。

Feign，声明式服务调用（HTTP客户端），本质上就是Ribbon+Hystrix

Stream，消息驱动，有Sink、Source、Processor三种通道，特性有订阅发布、消费组、消息分区（Redis,Rabbit、Kafka）。

Bus，消息总线，可与Spring Cloud Config联合实现热部署。

- **独自启动不需要依赖其它组件**

Eureka，服务注册中心，实现服务治理（服务注册与发现）。

Zuul，API服务网关，功能有路由分发和过滤。

Config，配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储，Git 以及 Subversion。

- **各个组件解决的问题**

Eureka和Ribbon，是最基础的组件，一个注册服务，一个消费服务。

Hystrix为了优化Ribbon、防止整个微服务架构因为某个服务节点的问题导致崩溃，是个保险丝的作用。

Dashboard给Hystrix统计和展示用的，而且监控服务节点的整体压力和健康情况。

Feign是方便我们程序员写更优美的代码的。

Zuul是加在整个微服务最前沿的防火墙和代理器，隐藏微服务结点IP端口信息，加强安全保护的。

Config是为了解决所有微服务各自维护各自的配置，设置一个统一的配置中心，方便修改配置的。

Bus是因为config修改完配置后各个结点都要refresh才能生效实在太麻烦，所以交给bus来通知服务节点刷新配置的。

Stream：是为了简化研发人员对MQ使用的复杂度，弱化MQ的差异性，达到程序和MQ松耦合。

集群/分布式/微服务/SOA 是什么？

网络编程

1.OSI七层模型

- 应用层
- 表示层
- 会话层
- 传输层 TCP UDP
- 网络层 IP
- 数据链路层
- 物理层

TCP/IP概念层模型：链路层、网络层（LVS负载均衡 IP）、传输层（TCP、UDP）、应用层 HTTP

2.一次完整的HTTP请求的完整过程

- ①建立 TCP 连接（之前可能还有一次DNS域名解析）
- ②客户端向服务器发送请求命令
- ③客户端发送请求头信息
- ④服务器服务器应答器
- ⑤返回响应头信息
- ⑥服务器向客户端发送数据
- ⑦服务器关闭 TCP 连接

3.RESTful

<https://blog.csdn.net/x541211190/article/details/81141459>

RESTful是目前最流行的 API 设计规范，用于 Web 数据接口的设计。

1.接口示例

- 传统URL请求格式：
 - <http://127.0.0.1/user/query/1> GET 根据用户id查询用户数据
 - <http://127.0.0.1/user/save> POST 新增用户
 - <http://127.0.0.1/user/update> POST 修改用户信息
 - <http://127.0.0.1/user/delete> GET/POST 删除用户信息
- RESTful请求格式：
 - <http://127.0.0.1/user/1> GET 根据用户id查询用户数据
 - <http://127.0.0.1/user> POST 新增用户
 - <http://127.0.0.1/user> PUT 修改用户信息
 - <http://127.0.0.1/user> DELETE 删除用户信息

1.TCP、UDP的区别？


1. TCP面向连接,提供可靠传输服务，保证数据准确性；UDP 是无连接的 尽最大努力交付，不保证可靠交付。

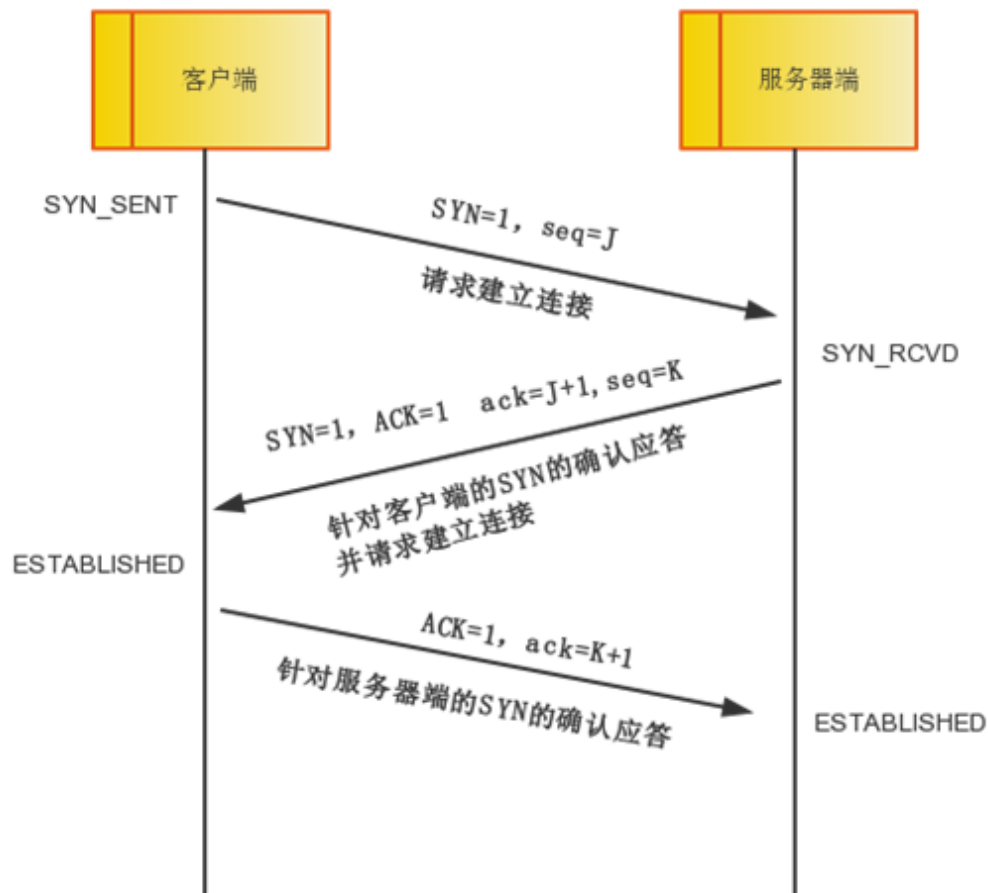
2. UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信。
3. 每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信。
4. TCP对系统资源要求较多，UDP对系统资源要求较少。


2、TCP协议如何保证可靠传输？

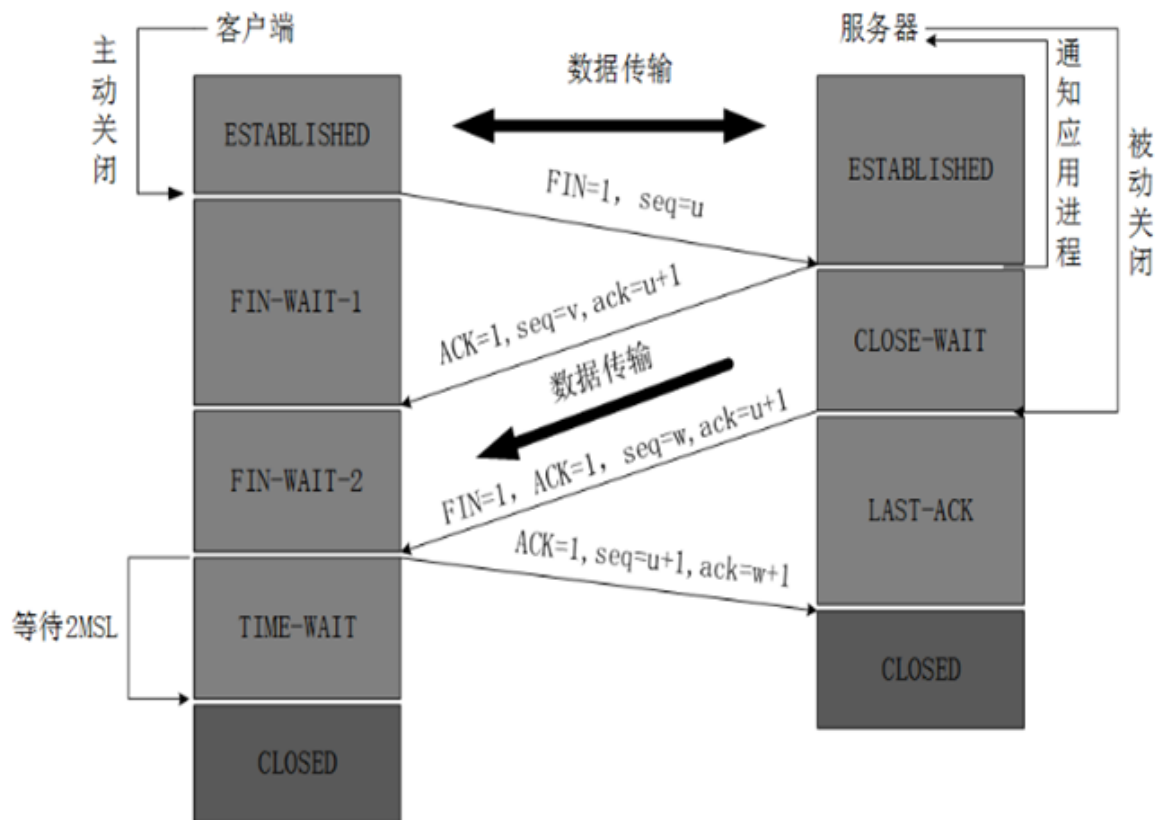
3、TCP的握手、挥手机制？

SYN ACK

 TCP三次握手



 TCP四次挥手



4.为什么Netty使用NIO而不是AIO?

原因: 在Linux系统上, AIO的底层实现仍使用EPOLL, 与NIO相同, 因此在性能上没有明显的优势; Windows的AIO底层实现良好, 但是Netty开发人员并没有把Windows作为主要使用平台考虑。

4、TCP的粘包/拆包原因及其解决方法是什么?

5、Netty的粘包/拆包是怎么处理的, 有哪些实现?

6、同步与异步、阻塞与非阻塞的区别?

7、说说网络IO模型?

8、BIO、NIO、AIO分别是什么?

9、select、poll、epoll的机制及其区别?

10、说说你对Netty的了解?

11、Netty跟Java NIO有什么不同, 为什么不直接使用JDK NIO类库?

12、Netty组件有哪些, 分别有什么关联?

13、说说Netty的执行流程?

14、Netty高性能体现在哪些方面?

15、Netty的线程模型是怎么样的?

16、Netty的零拷贝体现在哪里, 与操作系统上的有什么区别?

17、Netty的内存池是怎么实现的?

18、Netty的对象池是怎么实现的?

19、在实际项目中，你们是怎么使用Netty的？

20、使用过Netty遇到过什么问题？

21.浅谈Http和Https有什么区别

- HTTP是不安全的：数据拦截、数据篡改、数据攻击
HTTPS的安全需求：数据加密、身份验证、数据完整性
- 数据加密算法
对称加密：加解密只有一个密钥
非对称加密：公钥加密，私钥解密
- HTTPS比HTTP多出的事情：
 - 请求https连接获取证书（公钥）CA证书机构
 - 客户端给服务器发送公钥加密的随机数密文
 - 客户端同时给服务器发送公钥加密的随机数+私钥的密文
 - 服务器根据公钥解出随机数，同时解出私钥
 - 客户端使用非对称加密进行数据传输，客户端使用公钥加密，服务器使用私钥解密。

22.get和post请求的区别

get和post从实现本质上讲都是http->TCP协议，是没有区别的。但是GET产生一个TCP数据包；POST产生两个TCP数据包。

- 对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；
- 而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

他们的区别主要体现在以下几个方面：

- get请求用来从服务器上获得资源，而post是用来向服务器提交数据；
- GET参数通过URL传递，POST放在Request body中，因此POST比GET更安全，因为GET参数直接暴露在URL上，所以不能用来传递敏感信息。
- get传输的数据要受到URL长度限制（最大长度是 2048 个字符）；而post可以传输大量的数据，上传文件通常要使用post方式；
- GET请求只能进行url编码，而POST支持多种编码方式。

并发编程

1.启动线程的三种方式

- extends Thread
- runnable
- callable 可返回结果

2.run() start()

run方法就是普通对象的普通方法，不会启动线程；

只有调用了start()后，Java才会将线程对象和操作系统中实际的线程进行映射，再来执行run方法。

3.yield sleep wait

- yield: 是线程类 (Thread) 的静态方法, 它让掉当前线程 CPU 的时间片, 使正在运行中的线程重新变成就绪状态, 并重新竞争 CPU 的调度权。它可能会获取到, 也有可能被其他线程获取到。对象的机锁没有被释放。
- sleep: 方法是线程类 (Thread) 的静态方法, 让调用线程进入睡眠状态, 让出执行机会给其他线程, 等到休眠时间结束后, 线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法, 他不能改变对象的机锁, 当一个synchronized块中调用了sleep() 方法, 线程虽然进入休眠, 但是对象的机锁没有被释放, 其他线程依然无法访问这个对象。
- wait(): wait()是Object类的方法, 当一个线程执行到wait方法时, 它就进入到一个和该对象相关的等待池, 同时释放对象的机锁, 使得其他线程能够访问, 可以通过notify, notifyAll方法来唤醒等待的线程

4.join

线程A, 执行了线程B的join方法, 线程A必须要等待B执行完成了以后, 线程A才能继续自己的工作

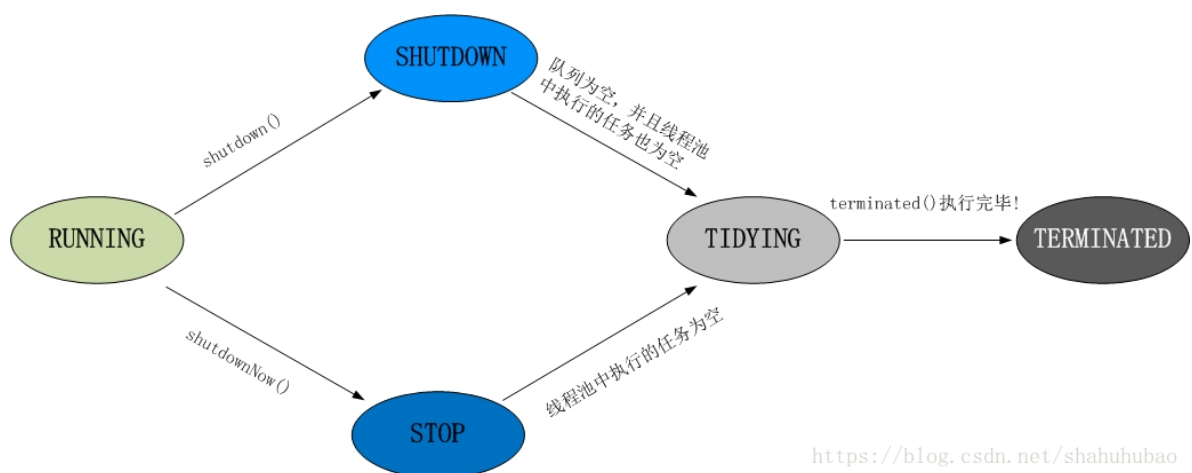
5.创建线程池有哪几种方式

- ①. newFixedThreadPool(int nThreads)
- ②. newCachedThreadPool()
- ③. newSingleThreadExecutor()
- ④. newScheduledThreadPool(int corePoolSize)

6.线程池都有哪些状态

线程池有5种状态: Running、ShutDown、Stop、Tidying、Terminated

 image



<https://blog.csdn.net/shahuhubao>

- **RUNNING**
 - (1) 状态说明: 线程池处在RUNNING状态时, 能够接收新任务, 以及对已添加的任务进行处理。
 - (2) 状态切换: 线程池的初始化状态是RUNNING。换句话说, 线程池被一旦被创建, 就处于RUNNING状态, 并且线程池中的任务数为0!
- **SHUTDOWN**

(1) 状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。

(2) 状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

- STOP

(1) 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。

(2) 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN) -> STOP。

- TIDYING

(1) 状态说明：当所有的任务已终止，ctl记录的“任务数量”为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

(2) 状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。

当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

- TERMINATED

(1) 状态说明：线程池彻底终止，就变成TERMINATED状态。

(2) 状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

7.线程池中 submit()和 execute()方法有什么区别？

- submit(Callable task)、submit(Runnable task, T result)、submit(Runnable task)归属于ExecutorService接口。
execute(Runnable command)归属于Executor接口。ExecutorService继承了Executor。
- submit有返回值，返回 Future，而execute没有
- submit()方便做异常处理。通过Future.get()可捕获异常。

8.在 java 程序中怎么保证多线程的运行安全

- 原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作，（atomic,synchronized）；
- 可见性：一个线程对主内存的修改可以及时地被其他线程看到，（synchronized,volatile）；
- 有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before原则）。

9.死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。是操作系统层面的一个错误，是进程死锁的简称。

10.怎么防止死锁

- 死锁的四个必要条件：
互斥条件：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源
- 请求和保持条件：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放

- 不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放
- 环路等待条件：是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防解除死锁。

11.ThreadLocal 是什么？有哪些使用场景？

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

12.说一下 synchronized 底层实现原理？

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象
- 同步方法块，锁是括号里面的对象

13.接口并行

```
//线程池
static ExecutorService taskExe = Executors.newFixedThreadPool(10);
@Override
public String getUserInfo(String useId) {
    long ti = System.currentTimeMillis();
    Callable<JSONObject> queryUserInfo = new Callable<JSONObject>() {
        @Override
        public JSONObject call() throws Exception {
            String userInfo = remoteService.getUserInfo(useId);
            JSONObject userJson = JSONObject.parseObject(userInfo);
            return userJson;
        }
    };

    Callable<JSONObject> queryMoneyInfo = new Callable<JSONObject>() {
        @Override
        public JSONObject call() throws Exception {
            String moneyInfo = remoteService.getUserMoney(useId);
            return JSONObject.parseObject(moneyInfo);
        }
    };

    FutureTask<JSONObject> userFutureTask = new FutureTask<>(queryUserInfo);
    FutureTask<JSONObject> moneyFutureTask = new FutureTask<>
(queryMoneyInfo);
```

```

//创建线程
taskExe.submit(userFutureTask); //http
taskExe.submit(moneyFutureTask);

JSONObject result = new JSONObject();
try {
    result.putAll(userFutureTask.get()); //http返回前，get停留在此
    result.putAll(moneyFutureTask.get());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}

return result.toString();
}

```

14.你是怎样控制缓存的更新

方案名	类型	实现思路
数据实时同步时效	增量/主动	强一致性，更新数据库后主动淘汰缓存，读请求更新缓存，为避免缓存雪崩，更新缓存的过程需要进行同步控制，同一时间只允许一个请求访问数据库，为了保证数据的一致性，还要加上缓存失效。
数据准实时更新	增量/被动	准一致性，更新数据库后，异步更新缓存，使用多线程技术或MQ实现。
任务调度更新	全量/被动	最终一致性，采用任务调度框架，按照一定频率更新。

绿色建筑大数据平台

单位时间内发电量统计（小时、日、月、年），发电、环境数据监测

监听redis，数据变化后，统计小时、日、月、年发电量，插入MySQL，淘汰缓存

客户端请求：优先查询缓存数据，缓存没有数据，查询MySQL数据

```

private final ConcurrentHashMap<String, ReentrantLock> locks = new
ConcurrentHashMap<>();

/**
 * 获取发电量
 *
 * @param userId
 * @return
 */
public BigDecimal getGenerationCapacity(String userId) {
    BigDecimal generationCapacity = null;
    //1.从缓存加载数据
    generationCapacity = getGenerationCapacityFromCache(userId);
    //2.缓存有数据，直接返回
    if (generationCapacity != null) {
        return generationCapacity;
    }
}

```

```

    }
    //竞争锁
    acquireLock(userId);
    try {
        generationCapacity = getGenerationCapacityFromCache(userId);
        //2.缓存有数据，直接返回
        if (generationCapacity != null) {
            return generationCapacity;
        }
        //3.如果没有数据，从MySQL中获取数据
        generationCapacity = getGenerationCapacityFromDB(userId);
        //4.数据库数据不为空，更新缓存
        if (generationCapacity != null) {
            upDateGenerationCapacityCache(userId);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        releaseLock(userId);
    }
    return generationCapacity;
}

private ReentrantLock getLockForKey(String userId){
    ReentrantLock lock = new ReentrantLock();
    //把新锁添加到locks中，如果成功（locks中不存在同一key的锁）使用新锁，如果失败使用
locks集合中的锁
    ReentrantLock previous = locks.putIfAbsent(userId, lock);
    return previous == null ? lock : previous;
}

/**
 * 从缓存获取数据
 *
 * @param userId
 * @return
 */
private BigDecimal getGenerationCapacityFromCache(String userId) {
    BigDecimal generationCapacity = null;
    return generationCapacity;
}

/**
 * 更新缓存
 *
 * @param userId
 * @return
 */
private void upDateGenerationCapacityCache(String userId) {

}

/**
 * 从DB获取数据
 *
 * @param userId
 * @return
 */

```

```

private BigDecimal getGenerationCapacityFromDB(String userId) {
    BigDecimal generationCapacity = null;
    return generationCapacity;
}

/**
 * 竞争锁
 *
 * @param userId
 */
private void acquireLock(String userId) {
    ReentrantLock lock = getLockForKey(userId);
    lock.lock();
}

/**
 * 释放锁
 *
 * @param userId
 */
private void releaseLock(String userId) {
    ReentrantLock lock = getLockForKey(userId);
    if(lock.isHeldByCurrentThread()){
        lock.unlock();
    }
}

```

15.调用第三方系统接口

1. 保证数据一致性：事务注解
 2. 性能优化：编程式事务
 3. CAS锁：状态机制 锁机制
- 锁或 synchronized 关键字可以实现原子操作，那么为什么还要用 CAS 呢，因为加锁或使用 synchronized 关键字带来的性能损耗较大，而用 CAS 可以实现乐观锁，它实际上是直接利用了 CPU 层面的指令，所以性能很高。

16.原子操作CAS

IO

1.java 中 IO 流分为几种

- 按功能来分：输入流（input）、输出流（output）。
- 按类型来分：字节流和字符流。

2.BIO,NIO,AIO 有什么区别

- **BIO (Blocking I/O):** 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。并发处理能力低，在活动连接数不是特别高（小于单机1000）的情况下使用。
- **NIO (New I/O):** NIO是一种同步非阻塞的I/O模型，对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发。

- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的。

集合

1.HashMap

1. 存储结构
数组、链表、红黑树 (JDK1.8)
2. 特点
快速存储、快速查找、可伸缩(负载因子0.75 扩容2倍)
负载因子较小：浪费存储空间
负载因子较大：哈希冲突概率增大，降低效率
3. hash算法
 $\text{hashCode} \wedge (\text{hashCode} \gg 16)$
4. 数组下标计算
 $\text{hash} \% 16$
5. 哈希冲突
不同对象计算出的数组下标相等。
使用单向链表解决哈希冲突，链表长度大于8，转红黑树

2.HashTable和ConcurrentHashMap实现线程安全

HashTable: 一把synchronized

ConcurrentHashMap: 分段锁 jdk1.8及以后节点锁

3.equals() 与 == 的区别是什么

- equals(): 判断两个对象是否相等
 - 类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
 - 类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。
- ==: 判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象

4.hashCode() 的作用是什么

hashCode() 的作用是**获取哈希码**，也称为散列码；它实际上是返回一个int整数。这个**哈希码的作用**是确定该对象在哈希表中的索引位置。 仅仅当创建并某个“类的散列表”(关于“散列表”见下面说明)时，该类的hashCode() 才有用。

5.hashCode() 和 equals() 之间有什么联系**

- **第一种 不创建“类对应的散列表”**
“hashCode() 和 equals() ”没有关系
- **第二种 创建“类对应的散列表”**

在这种情况下，该类的“hashCode() 和 equals() ”是有关系的：

1)、如果两个对象相等，那么它们的hashCode()值一定相同。

这里的相等是指，通过equals()比较两个对象时返回true。

2)、如果两个对象hashCode()相等，它们并不一定相等。

因为在散列表中，hashCode()相等，即两个键值对的哈希值相等。然而哈希值相等，并不一定能得出键值对相等。补充说一句：“两个不同的键值对，哈希值相等”，这就是哈希冲突。

hashCode()相等，通过equals()比较它们也返回true，认为对象相等。

6.Arrays.asList()使用指南

Arrays.asList() 方法返回的并不是 `java.util.ArrayList`，而是 `java.util.Arrays` 的一个内部类，这个内部类并没有实现集合的修改方法或者说并没有重写这些方法。

- 如何正确的将数组转换为ArrayList

- 自己动手实现

- 最简便的方法(推荐)

```
List list = new ArrayList<>(Arrays.asList("a", "b", "c"))
```

- 使用 Java8 的Stream(推荐)

```
Integer [] myArray = { 1, 2, 3 };
```

```
List myList = Arrays.stream(myArray).collect(Collectors.toList());
```

//基本类型也可以实现转换（依赖boxed的装箱操作）

```
int [] myArray2 = { 1, 2, 3 };
```

```
List myList = Arrays.stream(myArray2).boxed().collect(Collectors.toList());
```

- 使用 Guava(推荐)

- 于不可变集合，你可以使用 `ImmutableList` 类及其 `of()` 与 `copyOf()` 工厂方法：（参数不能为空）

```
List<String> i1 = ImmutableList.of("string", "elements"); // from
varargs
List<String> i1 = ImmutableList.copyOf(aStringArray);      // from
array
```

- 对于可变集合，你可以使用 `Lists` 类及其 `newArrayList()` 工厂方法

```
List l1 = Lists.newArrayList(anotherListOrCollection); // from collection
```

```
List l2 = Lists.newArrayList(aStringArray);           // from array
```

```
List l3 = Lists.newArrayList("or", "string", "elements"); // from varargs
```

7.Collection.toArray() 方法使用的坑&如何反转数组

- 该方法是一个泛型方法：`T[] toArray(T[] a)`；如果 `toArray` 方法中没有传递任何参数的话返回的是 `Object` 类型数组。

```
String [] s= new String[]{
    "dog", "lazy", "a", "over", "jumps", "fox", "brown", "quick", "A"
};
List<String> list = Arrays.asList(s);
Collections.reverse(list);
s=list.toArray(new String[0]); //没有指定类型的话会报错
```

由于JVM优化, `new String[0]` 作为 `Collection.toArray()` 方法的参数现在使用更好, `new String[0]` 就是起一个模板的作用, 指定了返回数组的类型, 0是为了节省空间, 因为它只是为了说明返回的类型。

8.Java集合框架常见问题

1. 说说List,Set,Map三者的区别

一、结构特点

List和Set是存储单列数据的集合, Map是存储键值对这样的双列数据的集合;

List中存储的数据是有顺序的, 并且值允许重复; Map中存储的数据是无序的, 它的键是不允许重复的, 但是值是允许重复的; Set中存储的数据是无顺序的, 并且不允许重复, 但元素在集合中的位置是由元素的hashCode决定, 即位置是固定的 (Set集合是根据hashCode来进行数据存储的, 所以位置是固定的, 但是这个位置不是用户可以控制的, 所以对于用户来说set中的元素还是无序的)。

二、实现类

List接口有三个实现类:

1.1 LinkedList

基于链表实现, 链表内存是散列的, 增删快, 查找慢;

1.2 ArrayList

基于数组实现, 非线程安全, 效率高, 增删慢, 查找快;

1.3 Vector

基于数组实现, 线程安全, 效率低, 增删慢, 查找慢;

Map接口有四个实现类:

2.1 HashMap

基于 hash 表的 Map 接口实现, 非线程安全, 高效, 支持 null 值和 null 键;

2.2 Hashtable

线程安全, 低效, 不支持 null 值和 null 键;

2.3 LinkedHashMap

是 HashMap 的一个子类, 保存了记录的插入顺序;

2.4 SortMap 接口

TreeMap, 能够把它保存的记录根据键排序, 默认是键值的升序排序

Set接口有两个实现类:

3.1 HashSet

底层是由 Hash Map 实现, 不允许集合中有重复的值, 使用该方式时需要重写 equals()和 hashCode()方法;

3.2 LinkedHashSet

继承于 HashSet, 同时又基于 LinkedHashMap 来进行实现, 底层使用的是 LinkedHashMap

三、区别

1. List 集合中对象按照索引位置排序, 可以有重复对象, 允许按照对象在集合中的索引位置检索对象, 例如通过list.get(i)方法来获取集合中的元素;
2. Map 中的每一个元素包含一个键和一个值, 成对出现, 键对象不可以重复, 值对象可以重复;
3. Set 集合中的对象不按照特定的方式排序, 并且没有重复对象, 但它的实现类能对集合中的对象按照特定的方式排序, 例如 Tree Set 类, 可以按照默认顺序, 也可以通过实现 Java.util.Comparator< Type >接口来自定义排序方式。

```

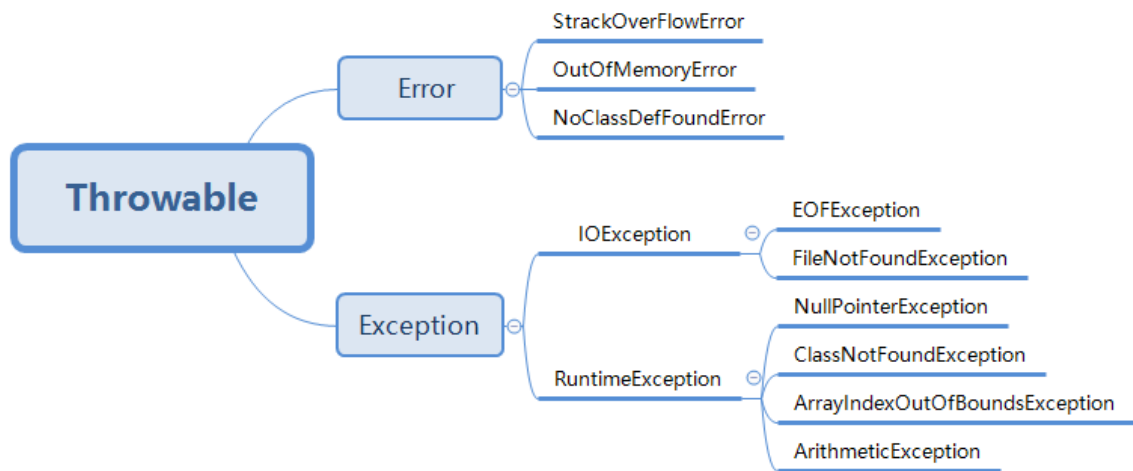
Set set = new HashSet();
Iterator it = set.iterator();
while (it.hasNext()) {
    String str = it.next();
    System.out.println(str);
}

```

2. ArrayList 与 LinkedList 区别

Java中的异常

image



JVM

1.Java内存区域

运行时数据区域

- **程序计数器**

程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。

- **虚拟机栈**

每个方法在被调用时都会创建一个栈帧，方法从调用到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

- **本地方法栈**

本地方法栈和虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为虚拟机使用到的Native方法服务。

- **Java堆**

是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，对象实例在这里分配内存。是GC管理的主要区域。

- **方法区**

方法区用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译期编译后的代码等数据。运行时常量池是方法区的一部分。

2.垃圾回收算法

- 标记-清除算法

将活跃对象进行标记，将未标记的对象清除。导致会存在很多内存碎片。

- 复制算法

标记存活对象，将存活对象复制到新的内存空间，清空原来空间。解决了内存碎片问题，缺点：内存浪费，总有一半的内存空间未被利用。

- 标记-整理算法

移动存活对象，一次性回收需要回收的区域。内存区域快整体移动，影响性能。

- 分代收集

新生代复制算法,老年代以标记整理算法为主。

3.类加载机制和双亲委派模型

- 类加载过程

- 加载

加载指的是将类的class文件读入到内存，并为之创建一个java.lang.Class对象。

- 连接：把类的二进制数据合并到RE中

1. 验证：检验被加载的类是否符合当前虚拟机的要求，有没有危害虚拟机安全的代码。

四种验证：

- 文件格式验证：主要验证字节流是否符合Class文件格式规范。
- 元数据验证：分析字节码描述的信息，是否符合java的语言语法的规范。
- 字节码验证：分析数据流和控制，确定语义是合法的，符合逻辑的。
- 符号引用验证：保证引用一定会被访问到，不会出现类等无法访问的问题。

2. 准备：为类的静态变量分配内存，并设置默认初始值。

3. 解析：将类的二进制数据中的符号引用替换成直接引用。

- 初始化：初始化是为类的静态变量赋予实际的初始值。

- 使用 new

- 卸载 GC

- 类加载器

- 启动类加载器：加载JAVA_HOME/lib目录下
- 扩展类加载器。加载JAVA_HOME/lib目录下
- 应用程序类加载器：加载用户路径上指定的类库

- 双亲委派模型

如果一个类加载器收到了类加载器的请求，首先加载任务委托给父类加载器，依次递归，直到启动类加载器。如果父类加载器能够完成加载任务，就成功返回；否则，子加载器才会尝试自己去加载。

双亲委派模型的优点：java类随着它的加载器一起具备了一种带有优先级的层次关系，保证java程序稳定运行。

虚拟机中决定一个类是否唯一：类本身和加载类的类加载器。

AOP, OOP

OOP

面向对象编程，是一种计算机编程架构。

针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。**竖向封装**

AOP

面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。

针对业务处理过程中的**切面**进行提取，它所面对的是处理过程中的某个**步骤或阶段**，以获得逻辑过程中各部分之间低耦合性的**隔离效果**。**横向封装**

```
@Aspect
public class MyAspect {

    /**
     * 前置通知
     */
    @Before("execution(* com.zejian.spring.springAop.dao.UserDao.addUser(..))")
    public void before(){
        System.out.println("前置通知....");
    }

    /**
     * 后置通知
     * returnVal, 切点方法执行后的返回值
     */
    @AfterReturning(value="execution(* com.zejian.spring.springAop.dao.UserDao.addUser(..)", returning = "returnVal")
    public void AfterReturning(Object returnVal){
        System.out.println("后置通知...."+returnVal);
    }

    /**
     * 环绕通知
     * @param joinPoint 可用于执行切点的类
     * @return
     * @throws Throwable
     */
    @Around("execution(* com.zejian.spring.springAop.dao.UserDao.addUser(..))")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("环绕通知前....");
        Object obj= (Object) joinPoint.proceed();
        System.out.println("环绕通知后....");
        return obj;
    }

    /**
     * 抛出通知
     * @param e
     */
}
```

```
    */
    @AfterThrowing(value="execution(*
com.zejian.spring.springAop.dao.UserDao.addUser(..)",throwing = "e")
    public void afterThrowable(Throwable e){
        System.out.println("出现异常:msg="+e.getMessage());
    }

    /**
     * 无论什么情况下都会执行的方法
     */
    @After(value="execution(*
com.zejian.spring.springAop.dao.UserDao.addUser(..)")
    public void after(){
        System.out.println("最终通知....");
    }
}
```