

JAVA集合

1、HashMap原理

1.1 rehash哈希冲突

哈希冲突指的是 在多个线程同时rehash同一个hashmap的过程中，可能会出现两个线程同时操作同一条链表产生链表节点之间相互应用的情况，此时在查询该链表时就会发生死循环。

解决方案：我们可以通过ConcurrentHashMap来解决。在多线程操作中，ConcurrentHashMap集合会对每一条链表都进行单独加锁，这样就只会有一条线程操作一条链表，避免了rehash哈希冲突，保证了线程安全。

1.2 HashMap怎么存储null键

当我们用put方法向HashMap中存入null键时，永远会放在第一个节点【主要是由于底层addEntry(hash, key, value, i)方法，当key为null，hash参数设置为0，默认添加到第一个节点】；

如果集合中已经存在null键，那么就会覆盖到已经存在的null键中的value值，返回值为被替代的value；如果不存在null键，则直接插入到数组第一个节点位置，返回null；

1.3 Hash算法

我们使用HashMap存储数据或者获取数据是调用put () 方法和get方法 () 都需要获取key键的hash值，而hash值的计算主要分为两步：

1、第一步：根据key键值 调用hashCode方法 获hashCode值

hashCode值得计算 是通过对象的内部物理地址转换成一个整数，然后该整数通过hash函数的算法（比如hash表中有八个位置，地址值转换为17， $17\%8=1$ ，那么该对象的hashCode值为1）得到一个hashCode值。

2、第二步 使用底层的hash方法 计算hash值【put和get方法】，是通过hashCode值 与自身无符号右移16位后做异或运算得到的结果，（做异或运算的目的是 能更好的保留各部分的特征，减少哈希碰撞）

```
//重新计算哈希值
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

扩展：获取hash值后 对槽位数-1进行取与运算 得到存储槽的位置；

hash算法不能被重写 因为被final关键词修饰

1.4 树化的条件

当长度大于8时，会由链表（函数 $n/2$ ）转为红黑树（ $\log(n)$ ），红黑树的查询效率更高；当小于6时，就不需要转化为红黑树；

2、树结构

B树：多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；

所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

B*树：在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3；

一棵m阶的B+树和m阶的B树的差异在于：

- 1、B+树有n棵子树的节点中含有n个关键字，B树是n棵子树有n-1个关键字。
- 2、B+树所有的叶子节点中包含全部关键字的信息，及指向这些关键字记录的指针，且叶子节点本身依关键字的大小自小而大的顺序连接。B树的叶子节点并没有包括全部需要查找的信息。
- 3、B+所有的非终端节点可以看成索引部分，节点中仅含有其子树根节点中最大（或最小）关键字。B树的非终结点也包含需要查找的有效信息。

list集合 数组 扩容机制（初始化，1.5倍） arraylist数组 linkedlist 双链表


set集合 单链表无序 treeset 红黑

hashmap集合 数据+单链表（长度是8红黑树）扩容（初始16，加载因子0.75 扩容2倍）

hashtable 安全 不接受null hashmap存取null 和哈希冲突

3、并发包

- 1、**CopyOnWriteArrayList** 安全的ArrayList 使用final修饰的lock锁 用的try finally释放

image-20200915092437380

- 2、**CopyOnWriteArraySet** 安全的hashSet

- 3、**ConcurrentHashMap** 安全的hashMap 锁单链

Hashtable是线程安全的，但效率低；

- 4、**CountDownLatch** 可以理解为多线程计算器，允许一个或多个线程等待其他线程完成操作

说明：

CountDownLatch中count down是倒数的意思，latch则是门闩的含义。整体含义可以理解为倒数的门栓，似乎有一点“三二一，芝麻开门”的感觉。CountDownLatch是通过一个计数器来实现的，每当一个线程完成了自己的任务后，可以调用countDown()方法让计数器-1，当计数器到达0时，调用CountDownLatch。

await()方法的线程阻塞状态解除，继续执行；

```
public CountDownLatch(int count)// 初始化一个指定计数器的CountDownLatch对象
public void await() throws InterruptedException// 让当前线程等待
public void countDown() // 计数器进行减1
```

5、CyclicBarrier

CyclicBarrier的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

```
public CyclicBarrier(int parties, Runnable barrierAction)// 用于在线程到达屏障时，优先执行 barrierAction，方便处理更复杂的业务场景
public int await()// 每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞
```

使用场景：CyclicBarrier可以用于多线程计算数据，最后合并计算结果的场景。需求：使用两个线程读取2个文件中的数据，当两个文件中的数据都读取完毕以后，进行数据的汇总操作。

6.Semaphore 并发数量控制

Semaphore可以设置同时允许几个线程执行。

Semaphore字面意思是信号量的意思，它的作用是控制访问特定资源的线程数目。

```
public void acquire() 表示获取许可    lock
public void release() 表示释放许可    unlock
```

7、线程信息交互_Exchanger

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。

两个线程通过exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

使用场景：可以做数据校对工作

需求：比如我们需要将纸制银行流水通过人工的方式录入成电子银行流水。为了避免错误，采用AB岗两人进行录入，录入到两个文件中，系统需要加载这两个文件，

并对两个文件数据进行校对，看看是否录入一致。

多线程

1、锁

锁的类型

1. **可重入锁**：在执行的对象中，所有同步方法不用再次获得锁。Synchronized和ReentrantLock都是可重入锁。
2. **可中断锁**：在等待获取过程中可中断。Synchronized内置锁是不可中断锁，ReentrantLock可以通过lockInterruptibly方法中断显性锁。
3. **公平锁**：按等待获取锁的进程获取，等待时间长的具有优先获取锁的权力。synchronized隐性锁是非公平锁，它无法保证等待的线程获取锁的顺序，ReentrantLock可以自己控制是否公平锁。
4. **读写锁**：对资源读取和写入的时候拆分为两部分处理，读的时候可以多线程一起读，写的时候必须同步写。ReadWriteLock是一个读写锁，通过readLock()获取读锁，通过writeLock()获取写锁。

synchronized锁：悲观锁 锁住代码块或者方法

lock锁：api lock() 加锁 以及unlock() 解锁 需要配合try/finally语句使用

两者区别：

- 1、synchronized是关键字，而Lock是一个接口。
- 2、lock锁通过方法加锁
- 3、lock锁需要手动释放，sync运行完自动释放
- 4、sync不可中断，除非抛出异常；Reentrantlock可以中断：设置超时方法trylock或者lockInterruptibly
- 5、sync非公平锁 ReentrantLock都可以，默认非公平锁
- 6、synchronized能锁住方法和代码块，而Lock只能锁住代码块。

多线程的开启方式

- 1、new thread() 继承Thread抽象类 开启
 - 2、实现 Runnable接口 放入new Thread中，start开启；
- start () 开启 jvm会调用本地方法（os的方法）去开启线程运行
- 如果直接运行run () 会在本线程直接运行该方法 不会开启多线程

多线程的关闭方法

其他线程中调用这个线程的打断方法 interrupt ()

循环监听自己的打断状态 会把多线程的打断标记改为true 就直接return 线程结束；

stop方法 是去调用底层系统函数去打断 会存在风险 无法保证代码的运行状态 可能多执行 可能少执行 还可能出错

线程的状态和声明周期：

新建new 运行 runnable 阻塞 block

等待状态waiting join () Object.wait () 此时必须等待另一个线程object.notify 唤醒（两个线程拿的必须是同一把锁）

超时等待状态 timed——wait sleep () 无限等待 除非自己设置时间

死亡状态terminated

Threadlocal ----- jdk提供的类

用来提供线程内部的共享变量，在多线程环境下，可以保证各个线程之间的变量互相隔离、相互独立。

ThreadLocal最简单的实现方式就是ThreadLocal类内部有一个线程安全的Map，然后用线程的ID作为Map的key，实例对象作为Map的value，这样就能达到各个线程的值隔离的效果。

拒绝策略：直接抛出异常 让当前线程执行 不用子线程 直接丢弃 队列中排列时间最长丢弃

接收多线程的返回值

Callable接口 里面重写call() 方法 负责执行返回 接口对象交给 ExecutorService的submit方法执行 Future负责接收

submit方法中传入callable接口 返回Future对象 调用future的get方法 获取返回值 get方式是同步的 我们可以设置等待时间 超时的话返回是null

2、线程池

- 1、线程池就是预先创建一些线程，它们的集合称为线程池。
- 2、线程池可以很好地提高性能，在系统启动时即创建大量空闲的线程，程序将一个task给到线程池，线程池就会启动一条线程来执行这个任务，执行结束后，该线程不会死亡，而是再次返回线程池中成为空闲状态，等待执行下一个任务。
- 3、线程的创建和销毁比较消耗时间，线程池可以避免这个问题。
- 4、Executors可以创建常见的4种线程（单线程池、固定大小的、可缓存的（可灵活回收空闲线程，若无可回收，则新建线程。））、可周期性执行任务的）。

1. newSingleThreadExecutor：创建一个单线程化的Executor。
2. newFixedThreadPool：创建一个固定大小的线程池。
3. newCachedThreadPool：创建一个可缓存的线程池
4. newScheduledThreadPool：创建一个定长的线程池，可以周期性执行任务。

阻塞队列

- ArrayBlockingQueue：基于数组实现的一个阻塞队列，在创建ArrayBlockingQueue对象时必须制定容量大小。并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列。
- LinkedBlockingQueue：基于链表实现的一个阻塞队列，在创建LinkedBlockingQueue对象时如果不指定容量大小，则默认大小为Integer.MAX_VALUE。
- SynchronousQueue：不储存元素的队列 用在newCached线程池中，来线程直接创建新的线程使用。
- PriorityBlockingQueue：以上2种队列都是先进先出队列，而PriorityBlockingQueue却不是，它会按照元素的优先级对元素进行排序，按照优先级顺序出队，每次出队的元素都是优先级最高的元素。注意，此阻塞队列为无界阻塞队列，即容量没有上限（通过源码就可以知道，它没有容器满的信号标志），前面2种都是有界队列。
- DelayQueue：基于PriorityQueue，一种延时阻塞队列，DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue也是一个无界队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

线程池：

newFixedThreadPool 固定大小的线程池 核心线程数固定

newCachedThreadPool 缓存线程池 sync队列 都是临时线程 无限创建和销毁

newSingleThreadExecutor 单线程线程池 保证所有任务按照顺序执行 如果异常死亡 重新创建一个新的顶替 补上

new ScheduledThreadPool 可以配置线程时间周期执行 主要是配置开始时间 和间隔时间

守护线程：

Java线程分为用户线程和守护线程。

守护线程是程序运行的时候在后台提供一种通用服务的线程。所有用户线程停止，进程会停掉所有守护线程，退出程序。

Java中把线程设置为守护线程的方法：在 start 线程之前调用线程的 setDaemon(true) 方法。

程序报出NPE为啥还能继续接受请求 线程池 线程隔离

单点登录系统

1、登录

用户访问系统1的受保护资源，系统1发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
sso认证中心发现用户未登录，将用户引导至登录页面
用户输入用户名密码提交登录申请
sso认证中心校验用户信息，创建用户与sso认证中心之间的会话，称为全局会话，同时创建授权令牌
sso认证中心带着令牌跳转会最初的请求地址（系统1）
系统1拿到令牌，去sso认证中心校验令牌是否有效
sso认证中心校验令牌，返回有效，注册系统1
系统1使用该令牌创建与用户的会话，称为局部会话，返回受保护资源
用户访问系统2的受保护资源
系统2发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
sso认证中心发现用户已登录，跳转回系统2的地址，并附上令牌
系统2拿到令牌，去sso认证中心校验令牌是否有效
sso认证中心校验令牌，返回有效，注册系统2
系统2使用该令牌创建与用户的局部会话，返回受保护资源

2、注销

sso认证中心一直监听全局会话的状态，一旦全局会话销毁，监听器将通知所有注册系统执行注销操作

用户向系统1发起注销请求
系统1根据用户与系统1建立的会话id拿到令牌，向sso认证中心发起注销请求
sso认证中心校验令牌有效，销毁全局会话，同时取出所有用此令牌注册的系统地址
sso认证中心向所有注册系统发起注销请求
各注册系统接收sso认证中心的注销请求，销毁局部会话
sso认证中心引导用户至登录页面

<https://www.cnblogs.com/markleilei/p/6201665.html>

SDK充值系统



我们主要用到微信支付SDK的以下功能

```
//获取随机字符串
wxPayUtil.generateNonceStr();
//Map转换为XML字符串（自动添加签名）
wxPayUtil.generateSignedXml(param, partnerkey);
//XML字符串转换为MAP
wxPayUtil.xmlToMap(result);
```

```
#微信支付信息配置 app唯一标识 商户账号 商户秘钥 回调地址
weixin:
  appid: wx8397f8696b538317
  partner: 1473426802
  partnerkey: T6m9iK73b0kn9g5v426MKfHQH7X8rKwb
  notifyurl: http://www.itcast.cn
```

Spring + SSM

1、springmvc的工作流程

 image-20200831091104685

处理器映射器、处理器适配器、视图解析器称为 SpringMVC 的三大组件。

2、Spring的特点

- 1、**轻量级**：完整的spring文件大小和开销都不大
- 2、**控制反转**：IOC技术 促进了低耦合
- 3、**面向切面**：AOP 应用业务逻辑和系统服务分开
- 4、**容器**：管理引用对象的配置和生命周期
- 5、**框架**：将简单的组件配置 提供事务管理功能 将逻辑开发留给开发者

3、Spring的三大组件

Bean组件：解决三件事——Bean的创建、Bean的定义、Bean的解析；

Bean的创建：通过工厂模式实现 顶层接口：BeanFactory

Bean的定义：在Spring配置文件节点中，包括子节点等；Spring内部它被转换为BeanDefinition对象；

Bean的解析：DefinitionReader 就是对配置文件以及对Tag的解析

Context组件：作为Spring的IOC容器（应用加载时创建的Map集合称之为容器）1、标识了一个应用环境；2、利用BeanFactory创建Bean对象；3、保存对象关系表；4、能够捕获各种事件

Core组件：访问资源 Resource接口封装了各种可能的资源类型，继承了InputStreamSource接口。

4、BeanFactory 创建对象

程序初始化时，解析xml（读xml流，生产Document对象 再通过Document获取所有的bean标签 遍历所有的bean标签 获取id和class值 反射创建所有Bean的map对象存入beanMap中）


```

static {
    InputStream is = null;
    try {
        //1.创建SAXReader对象 使用DOM4J解析spring.xml文件，根据id获取类的全限定名
        SAXReader saxReader = new SAXReader();
        //2.读取xml 获得document对象
        is =
        BeanFactoryOri.class.getClassLoader().getResourceAsStream("applicationContext.xml");

        Document document = saxReader.read(is);
        //3.获得所有的bean标签
        List<Element> beanEles = document.selectNodes("//bean");

        //4.遍历所有的bean标签
        for (Element beanEle : beanEles) {
            //5.获得bean标签的id和class属性值
            String id = beanEle.attributeValue("id");
            String className = beanEle.attributeValue("class");

            //6.根据class属性值反射创建对象，把id作为key,把反射创建的对象作为value存
            到beanMap

            Object obj = Class.forName(className).newInstance();
            beanMap.put(id,obj);
        }
    }
}

```

配置结构

```
<bean id/name="" class="" scope="" init-method="" destroy-method=""></bean>
```

- id/name属性
用于标识bean，其实id和name都必须具备唯一标识，两种用哪一种都可以。但是一定要唯一、一般开发中使用id来声明。
- class属性: 用来配置要实现化类的全限定名
- scope属性: 用来描述bean的作用范围
singleton: 默认值，单例模式。spring创建bean对象时会以单例方式创建。(默认)
prototype: 多例模式。spring创建bean对象时会以多例模式创建。
request: 针对Web应用。spring创建对象时，会将此对象存储到request作用域。
session: 针对Web应用。spring创建对象时，会将此对象存储到session作用域。
- init-method属性: spring为bean初始化提供的回调方法
- destroy-method属性: spring为bean销毁时提供的回调方法. 销毁方法针对的都是单例bean，如果想销毁bean，可以关闭工厂

获取Bean对象

6、实例化Bean的三种方式

1. 无参构造方法方式 直接xml配置

```

<bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl"></bean>

```


2. 静态工厂方式 提供静态工厂类 new 实例化对象

```
public class StaticFactory {  
    public static Object getBean(){  
        return new AccountServiceImpl02();  
    }  
}
```

```
<!--静态工厂方式 -->  
<bean id="accountService02" class="com.itheima.utils.StaticFactory"  
factory-method="getBean"/>
```

3. 实例工厂实例化的方法

```
public class InstanceFactory {  
    public Object getBean(){  
        return new AccountServiceImpl03();  
    }  
}
```

```
<!-- 方式三：实例化工厂 -->  
<!--注册工厂 -->  
<bean id="factory" class="com.itheima.utils.InstanceFactory"></bean>  
<!--引用工厂 -->  
<bean id="accountService03" factory-bean="factory" factory-method="getBean"/>
```

7、Spring依赖注入DI 的方式

1、set方法注入 2、构造器注入 3、静态工厂的方法注入 4、实例工厂的方法注入

1,2支持注解+xml方式 (@Autowired @Resource @Required)

3,4只能使用配置文件xml方式

1、set+注解

```
//注解注入（autowire注解默认使用类型注入）  
@Autowired //【看这里】  
private UserDao userDao;
```

2、构造+注解 要注入的对象

```
public class UserService {  
    private UserDao userDao;  
    //注解到构造方法处  
    @Autowired //【看这里】  
    public UserService(UserDao userDao) {  
        this.userDao = userDao;  
    }  
}
```

3、set+xml property标签

```
<bean name="userService" class="com.obob.service.UserService">
  <property name="userDao" ref="userDao" /><!--这里是property-->
</bean>
<bean name="userDao" class="com.obob.dao.UserDao"></bean>
```

4、构造+xml constructor-arg标签

```
<bean name="userService" class="com.obob.service.UserService">
  <constructor-arg index="0" ref="userDao"></constructor-arg> <!--这里是
constructor-arg-->
</bean>
<bean name="userDao" class="com.obob.dao.UserDao"></bean>
```

5、静态工厂+xml property+ref

```
<bean name="userService" class="com.obob.service.UserService">
  <property name="staticUserDao" ref="staticUserDao" /><!--property属性-->
</bean>
<!--UserDao staticUserDao=Factory.initUserDao() -->
<bean name="staticUserDao" class="com.obob.Factory" factory-
method="initUserDao"></bean>
```

6、实例工厂+xml property+factory-bean

```
<bean name="userService" class="com.obob.service.UserService">
  <property name="staticUserDao" ref="staticUserDao" />
</bean>
<!--UserDao staticUserDao=factory.initUserDao() -->
<bean name="staticUserDao" factory-bean="factory" factory-method="initUserDao">
</bean>

<!--Factory factory = new Factory() -->
<bean name="factory" class="com.obob.Factory"></bean>
```

8、Spring事务管理

Spring管理事务是通过事务管理器. 定义了一个接口PlatformTransactionManager

read-only: 是否是只读事务。默认false, 不只读。

isolation: 指定事务的隔离级别。默认值是使用数据库的默认隔离级别。

propagation: 指定事务的传播行为。

timeout: 指定超时时间。默认值为: -1。永不超时。

rollback-for: 用于指定一个异常, 当执行产生该异常时, 事务回滚。产生其他异常, 事务不回滚。没有默认值, 任何异常都回滚。

no-rollback-for: 用于指定一个异常, 当产生该异常时, 事务不回滚, 产生其他异常时, 事务回滚。没有默认值, 任何异常都回滚。

```
@Transactional(propagation=Propagation.REQUIRES_NEW,  
isolation=Isolation.READ_COMMITTED,  
noRollbackFor={UserAccountException.class},  
rollbackFor = IOException.class,  
readOnly=false,  
timeout=-1)
```

事务的传播行为的取值

保证在同一个事务里面:

- **PROPAGATION_REQUIRED:默认值, 也是最常用的场景.**

如果当前没有事务, 就新建一个事务,
如果已经存在一个事务中, 加入到这个事务中。

- **PROPAGATION_SUPPORTS:**

如果当前没有事务, 就以非事务方式执行。
如果已经存在一个事务中, 加入到这个事务中。

- **PROPAGATION_MANDATORY**

如果当前没有有事务, 就抛出异常;
如果已经存在一个事务中, 加入到这个事务中。

保证不在同一个事物里:

- **PROPAGATION_REQUIRES_NEW**

如果当前有事务, 把当前事务挂起,创建新的事务但独自执行

- **PROPAGATION_NOT_SUPPORTED**

如果当前存在事务, 就把当前事务挂起。不创建事务

- **PROPAGATION_NEVER**

如果当前存在事务, 抛出异常



其他

- 1、事务方法的嵌套调用会产生事务传播。
- 2、spring的事务管理是线程安全的。
- 3、父类的声明@Transactional会对子类的所有方法进行事务增强, 子类覆盖重写父类方式可以覆盖其@Transactional中的声明配置
- 4、类名上方使用@Transactional, 类中方法可通过属性配置来覆盖类上的 @Transactional 配置

Transactional事务失效的原因

- 1、数据库需要有事务, 比如myISAM不支持事务
- 2、spring必须得注入dao 同样是因为基于数据库 (可不说)

- 3、普通方法调用有事务的方法 有事务的方法会失效（因为aop机制不会拦截内部函数间的调用）
- 4、函数必须是public 否则不生效
- 5、因为Spring的事务基于AOP 所以该类必须托管于Spring容器内 且必须与调用者处在同一个容器内部 否则不生效
- 6、开启注解驱动（boot自动开启）

9、spring整合mybatis

1. 引入mybatis以及整合的相关依赖
2. 在spring的IOC容器中创建DataSource对象(可以使用spring内置的DataSource)
3. 在spring的IOC容器中创建SqlSessionFactoryBean对象，并且可以指定要配置别名的包或者是加载核心配置文件
4. 在spring的IOC容器中创建MapperScannerConfigurer对象，用于扫描Dao接口创建代理对象，创建代理对象的目的是执行SQL语句
5. 通过注解的方式注入Dao代理对象

10、spring整合JUnit

1. 添加Spring整合单元测试的坐标
2. 在测试类上面添加注解
 - @RunWith(SpringJUnit4ClassRunner.class) 指定运行的环境
 - @ContextConfiguration() 加载配置文件或者配置类的

11、AOP的底层 两种代理方式

AOP 的底层动态代理实现有两种方案。

一种是使用JDK的动态代理。这一种主要是针对有接口实现的情况。它的底层是创建接口的实现代理类，实现扩展功能。也就是我们要增强的这个类，实现了某个接口，那么我就可以使用这种方式了。

另一种方式是使用了cglib 的动态代理，这种主要是针对没有接口的方式，那么它的底层是创建被目标类的子类，实现扩展功能。

JDK默认方式

获取代理对象: Proxy.newProxyInstance(类加载器，所有接口，重写invoke方法的InvocationHandler的对象)

```
public class B_JDKProxy {

    @Test
    public void fun01(){
        AccountDao accountDao = new AccountDaoImpl();

        AccountDao proxyDao = (AccountDao)
        Proxy.newProxyInstance(accountDao.getClass().getClassLoader(),
        accountDao.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
                if("save".equals(method.getName())){
                    System.out.println("权限校验...");
                }
            }
        });
    }
}
```

```

        return method.invoke(accountDao, args);
    }
    return method.invoke(accountDao, args);
}
});
proxyDao.save();
}
}

```

CGLIB代理方式

第三方的代理机制，不是jdk自带的。没有实现接口的类产生代理，使用的是字节码的增强技术，其实就是产生这个类的子类。

获取代理对象：Enhancer对象。creat () 方法

```

public class C_CglibProxy {
    @Test
    public void fun01(){
        AccountDaoImpl accountDao = new AccountDaoImpl();

        //创建enhancer对象
        Enhancer enhancer = new Enhancer();
        //设置代理的父类
        enhancer.setSuperclass(AccountDaoImpl.class);

        enhancer.setCallback(new MethodInterceptor() {

            @Override
            public Object intercept(Object obj, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
                if("save".equals(method.getName())){
                    System.out.println("权限校验...");
                    return method.invoke(accountDao, args);
                }
                return method.invoke(accountDao, args);
            }
        });

        AccountDaoImpl accountDaoProxy = (AccountDaoImpl) enhancer.create();
        accountDaoProxy.save();
    }
}

```

image-20200831210704290

12、基于注解的AOP配置

1. 创建工程, 导入坐标
2. 创建Dao和增强的类, 注册
3. 开启AOP的注解支持
4. 在增强类上面添加@Aspect
5. 在增强类的增强方法上面添加
 - o @Before()
 - o @AfterReturning()

- @Around()
- @AfterThrowing()
- @After()

```
@Component
@Aspect
public class MyAspect {
    @Pointcut("execution(* com.itheima.service.impl.UserServiceImpl.*(..))")
    public void pt1(){

    }

    @Pointcut("execution(* com.itheima.service.impl.UserServiceImpl.add(..))")
    public void pt2(){

    }

    /**
     * 权限校验
     */
    @Before("pt1()")
    public void checkPermission(){
        System.out.println("进行权限校验...");
    }

    /**
     * 打印日志
     */
    @AfterReturning("pt1()")
    public void printLog(){
        System.out.println("打印日志...");
    }

    /**
     * 友好通知
     */
    @AfterThrowing("pt1()")
    public void say(){
        System.out.println("服务器出现异常，请稍后...");
    }

    @After("pt1()")
    public void end(){
        System.out.println("执行完毕....");
    }

    /**
     * 计算执行时间
     * @param joinPoint
     * @throws Throwable
     */
    @Around("pt2()")
    public void time(ProceedingJoinPoint joinPoint) throws Throwable {
        long startTime = System.currentTimeMillis();
        //执行被增强的那个方法
        joinPoint.proceed();
        long endTime = System.currentTimeMillis();
        System.out.println(endTime - startTime);
    }
}
```

```
}  
}
```

aop实现方式

13、SpringMVC

1、@RequestMapping的属性

path: 指定请求路径的url

value: =path

method: 指定该方法的请求方式

params: 指定限制请求参数的条件, 不是会报错

headers: 发送的请求头中必须包含的请求头

```
//请求参数必须是money=18,如果不是,则会报错(HTTP Status 400 - Bad Request)  
@RequestMapping(value = "/add",params = {"money=18"})
```

2、接收参数注解

1. 请求参数类型是简单(基本,String)类型
 - 方法的形参和请求参数的name一致就可以
2. 请求参数类型是pojo对象类型
 - 形参就写pojo对象
 - pojo的属性必须和请求参数的name一致就可以
3. 获取参数直接封装到List或者Map中的时候, 要加上RequestParam注解

3、请求参数接收细节

3.1 请求参数乱码处理 --- 在web.xml里面配置过滤器

```
<!-- 配置spring提供的字符集过滤器 -->  
<filter>  
    <filter-name>CharacterEncodingFilter</filter-name>  
    <filter-  
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  
    <!-- 配置初始化参数, 指定字符集 -->  
    <init-param>  
        <param-name>encoding</param-name>  
        <param-value>UTF-8</param-value>  
    </init-param>  
</filter>  
<filter-mapping>  
    <filter-name>CharacterEncodingFilter</filter-name>  
    <url-pattern>/*</url-pattern>  
</filter-mapping>
```

3.2 使用DateTimeFormate注解进行局部类型转换

默认情况下, SpringMVC已经实现一些数据类型自动转换。 内置转换器全都在:

`org.springframework.core.convert.support` 包下, 如遇特殊类型转换要求, 需要我们自己编写自定义类型转换器。

因为SpringMVC内置的是将 1999/03/04这种格式的字符串转换成Date，如果客户端传入的是1999-03-04这种格式的，所以无法进行转换，才报错了；

在要进行转换的变量上添加DateTimeFormat注解，指定转换的格式

```
@DateTimeFormat("yyyy-MM-dd")
private Date birthday;
```

这种方式有局限性，只能是添加了DateTimeFormat注解的变量能够进行转换，没有添加的变量还是会使用SpringMVC默认转换规则

14、springmvc的常用注解

@RequestParam 请求中名称不一致参数赋值

@RequestBody post请求参数转换为对象 必须加

@PathVariable 获取RestFul风格的url上的参数

restful风格编程：1、uri指向一个特定的资源 2、请求方式转换为基本操作的状态转换 getmapping 获取改资源

属性：

value：用于指定 url 中占位符名称。

required：是否必须提供占位符。

```
@RequestMapping(value = "info/{id}",method = RequestMethod.POST)
public String addInfo(@PathVariable("id") int id){
    System.out.println("添加信息:" + id);
    return "success";
}
```

@RequestHeader：获取请求消息的请求头（提供消息头名称）

属性：

value：提供消息头名称

required：是否必须有此消息头

```
@RequestMapping("testRequestHeader")
public String testRequestHeader(@RequestHeader(value = "User-Agent") String
requestHeader){
    System.out.println("requestHeader="+requestHeader);
    return "success";
}
```

@CookieValue：获取指定CookieValue

属性：

value：指定 cookie 的名称。

required：是否必须有此 cookie。

```

@RequestMapping("testCookieValue")
public String testCookieValue(@CookieValue(value="JSESSIONID") String sessionId)
{
    System.out.println("sessionId="+sessionId);
    return "success";
}

```

@ResponseBody: 响应json数据

该注解用于将 Controller 的方法返回的对象，通过 `HttpMessageConverter` 接口转换为指定格式的数据如：json,xml 等，通过 Response 响应给客户端

4、转发和重定向

forward转发

controller 方法在提供了 String 类型的返回值之后，默认就是请求转发。我们也可以加上 `forward:` 可以转发到页面,也可以转发到其它的controller方法

redirect方法重定向到页面：它相当于“`response.sendRedirect(url)`”。需要注意的是，如果是重定向到jsp 页面，则jsp 页面不能写在 WEB-INF 目录中，否则无法找到。

转发到页面

`forward:/页面的路径`

2. 转发到Controller

`forward:/类上面的RequestMapping/方法上面的RequestMapping`

3. 重定向到页面

`redirect:/页面的路径`

4. 重定向到Controller

`redirect:/类上面的RequestMapping/方法上面的RequestMapping`

14、SpringMVC的异常处理

自定义异常类 继承exception

```

/**
 * 自定义异常类
 */
public class SysException extends Exception{}

```

自定义异常处理器 实现异常处理器 `HandlerExceptionResolver`

```

/**
 * 自定义异常处理器
 */
public class SysExceptionHandler implements HandlerExceptionResolver

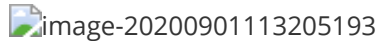
```

15、SpringMVC的拦截器

自定义拦截器 1、写拦截类方法 2、在配置文件中设置拦截器路径

必须实现拦截处理器接口 HandlerInterceptor接口

1. 拦截器的放行



2. 拦截后跳转

拦截器的处理结果，莫过于两种：

放行： 如果后面还有拦截器就执行下一个拦截器，如果后面没有了拦截器，就执行Controller方法

拦截： 但是注意，拦截后也需要返回到一个具体的结果(页面,Controller)。

在preHandle方法返回false,通过request进行转发,或者通过response对象进行重定向,输出

```
public class Interceptor01 implements HandlerInterceptor {
    @Override
    //在达到目标方法之前执行(拦截的方法)
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("preHandle 执行了...");
        //转发到拦截后的页面

        //request.getRequestDispatcher("/interceptor01.jsp").forward(request, respon
se);

        //转发到controller

        request.getRequestDispatcher("/demo01/fun02").forward(request, response);
        return false; //返回true放行， 返回false拦截
    }
}
```

3. 拦截器的路径



5. 拦截器的其它方法

- afterCompletion 在目标方法完成视图层渲染后执行。
- postHandle 在目标方法执行完毕获得了返回值后执行。
- preHandle 被拦截的目标方法执行之前执行。

```
public class Interceptor01 implements HandlerInterceptor {
    @Override
    //在达到目标方法之前执行(拦截的方法)
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("preHandle 执行了...");
        return true; //返回true放行， 返回false拦截
    }
    @Override
    //在目标方法执行完成之后,完成页面渲染之前执行
    public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, @Nullable ModelAndView
modelAndView) throws Exception {
    }
}
```

```

        System.out.println("postHandle 执行了...");
    }

    @Override
    //完成页面渲染之后执行
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, @Nullable Exception ex)
        throws Exception {
        System.out.println("afterCompletion 执行了...");
    }
}

```

6. 多个拦截器执行顺序

如果采用配置文件方式配置过滤器，那么就**按照过滤器的配置先后顺序执行**

如果采用注解方式配置过滤器，那么就**按照类名的排序执行**

```

<!--配置拦截器-->
<mvc:interceptors>
    <mvc:interceptor>
        <!--用于指定拦截的路径-->
        <mvc:mapping path="/*" />
        <!--用于指定忽略(不拦截的路径)-->
        <mvc:exclude-mapping path="/demo01/fun02" />
        <bean id="interceptor01"
class="com.itheima.web.interceptor.Interceptor01"></bean>
    </mvc:interceptor>
    <mvc:interceptor>
        <!--用于指定拦截的路径-->
        <mvc:mapping path="/*" />
        <!--用于指定忽略(不拦截的路径)-->
        <mvc:exclude-mapping path="/demo01/fun02" />
        <bean id="interceptor02"
class="com.itheima.web.interceptor.Interceptor02"></bean>
    </mvc:interceptor>
</mvc:interceptors>

```

Servlet

1、servlet的生命周期

初始化：web容器加载servlet，调用init () 方法

处理请求：当请求到达时，运行期service () 方法，

销毁：服务结束 web容器调用servlet的destory () 方法销毁

2、doGet与doPost方法的两个参数是什么

HttpServletRequest：封装了与请求相关的信息

HttpServletResponse：封装了与响应相关的信息

SpringBoot

1、Maven的聚合和继承

- 聚合可以一起构建多个module，pom.xml中的体现主要是和
- 继承的特性是指建立一个父模块，我们项目中的多个模块都做为该模块的子模块，将各个子模块相同的依赖和插件配置提取出来，从而简化配置文件，**父模块的打包方式必须为pom，否则无法构建项目。** pom.xml中的体现就是结点

2、Springboot整合MyBatis

#2.添加起步依赖 自身的web依赖+mabatis依赖+mysql连接驱动依赖

3.创建POJO

4.创建mapper接口

5.创建映射文件

6.配置yml 指定映射文件位置

7.创建启动类，加入注解扫描

8.创建service controller 进行测试

```
@MapperScan(basePackages = "com.itheima.dao")
```

//MapperScan 用于扫描指定包下的所有的接口，将接口产生代理对象交给spring容器

3、SpringBoot版本控制的原理

底层中spring-boot-dependencies中对各个版本进行了定义 我们的spring-boot-starter-parent继承了dependencies，parent子项目又继承了parent；通过maven的依赖传递特性 控制了版本的统一管理。

4、SpringBoot的自动配置原理

4.1 Condition接口 关联的注解为@conditional结合起来使用。

实现条件判断功能，用于选择性的创建Bean对象到spring容器中。

实现condition接口 重写里面的matches方法 写业务逻辑 通过Class.forName() 去加载字节码文件 加载成功说明有 return true，失败则false

在org.springframework.boot.autoconfigure.condition 包内有多个注解

ConditionalOnBean 当spring容器中有某一个bean时使用

ConditionalOnClass 当判断当前类路径下有某一个类时使用

ConditionalOnMissingBean 当spring容器中没有某一个bean时才使用

ConditionalOnMissingClass 当当前类路径下没有某一个类的时候才使用

ConditionalOnProperty 当配置文件中有一个key value的时候才使用

....

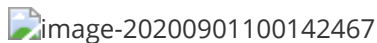
4.2Enable*类型注解

@SpringBootApplication注解里面有@EnableAutoConfiguration，那么这种@Enable*开头就是springboot中定义的一些动态启用某些功能的注解，他的底层实现原理实际上用的就是@import注解导入一些配置，自动进行配置，加载Bean。

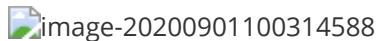
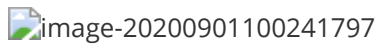
在启动类的注解@springbootapplication注解里面又修饰了@compnetScan注解，该注解的作用用于组件扫描包类似于xml中的context-componet-scan，如果不指定扫描路径，那么就扫描该注解修饰的启动类所在的包以及子包。这就是为什么我们在第一天的时候写了controller 并没有扫描也能使用的原因。

4.3自动配置流程

- 1.@import注解 导入配置
- 2.selectImports导入类中的方法中加载配置返回Bean定义的字符数组
- 3.加载META-INF/spring.factories 中获取Bean定义的全路径名返回
- 4.最终返回回去即可



过程： springboot启动类启动时，会加载springbootapplication配置文件，里面配置了@EnableAutoConfiguration 和@ComponentScan 注解；1、Auto自动配置文件中使用@import注解导入了AutoConfigarationImportSelector.class 配置文件 2、selectImports 导入类中方法上的加载配置，返回Bean定义的字符数组；3、加载META-INF/spring.factories 中获取Bean定义的全路径名返回；



4.3修改web容器：

如上，我们可以通过修改web容器，根据业务需求使用性能更优越的等等其他的web容器。这里我们演示使用jetty作为web容器。

在pom.xml中排除tomcat依赖 添加jetty依赖

在spring-boot-starter-web依赖里面 写标签 排除spring-boot-starter-tomcat依赖；再添加spring-boot-jetty依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>spring-boot-starter-tomcat</artifactId>
      <groupId>org.springframework.boot</groupId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

4.4、监控SpringBoot的组件-Actuator

时常我们在使用的项目的时候，想知道相关项目的一些参数和调用状态，而SpringBoot自带监控功能Actuator，可以帮助实现对程序内部运行情况监控，比如监控状况、Bean加载情况、配置属性、日志信息等。

1. 创建springboot工程
2. 添加Actuator的起步依赖
3. 配置开启端点和相关配置项
4. 通过端点路径查看信息

4.5、SpringBoot部署项目

在springboot项目中，我们部署项目有两种方式：

- jar包直接通过java命令运行执行 package命名打成jar包 执行.jar命令
- war包存储在tomcat等servlet容器中执行

package打成war包，放置在tomcat的webapps里面

注意：得首先修改启动类配置 需要继承SpringBootServletInitializer

再执行package命令打包

4.6、SpringBoot监听机制

springboot事件监听机制，实际上是对java的事件的封装。

java中定义了以下几个角色：

- 事件 Event
- 事件源 Source
- 监听器 Listener 实现EventListener接口的对象

ApplicationContextInitializer
SpringApplicationRunListener
CommandLineRunner
ApplicationRunner

解释：

CommandLineRunner 在容器准备好了之后可以回调 @componet修饰即可
ApplicationRunner 在容器准备好了之后可以回调 @componet修饰即可

ApplicationContextInitializer

在spring 在刷新之前 调用该方法 用于：做些初始化工作 通常用于web环境，用于激活配置，web上下文的属性注册。

注意：需要配置META/spring.factories 配置之后才能加载调用

SpringApplicationRunListener 也需要配置META/spring.factories 配置之后才能加载调用

他是SpringApplication run方法的监听器，当我们使用SpringApplication调用Run方法的时候触发该监听器回调方法。注意：他需要有一个公共的构造函数，并且每一次RUN的时候都需要重新创建实例

4.7、SpringBoot中启动类的启动


1、springapplication（简称app）启动，先构造一个Spring应用，new SpringApplication; 该对象会进行初始化，包括配置source、配置是否是web环境 创建初始化构造器、创建应用监听器等；

创建初始化构造器（工厂）对象 使用springboot的自动配置 生成工厂类实例返回



创建完这个对象 执行该app.run();

应用启动计时器 和监听器开始监听 配置环境模块（ConfigurableEnvironment）、

image-20200901131645165


image-20200901110312944

image-20200901131816386

<https://www.processon.com/view/link/59812124e4b0de2518b32b6e>

SpringCloud

能够理解SpringCloud作用

- 用来做微服务架构的技术解决方案
- SpringCloud基于SpringBoot开发的，SpringCloud整合了很多优秀的第三方微服务开源框架

RPC和HTTP

RPC： 基于远程过程调用 底层是基于Socket 是根据API来定义

速度快，效率高。 典型代表Dubbo WebService ElasticSearch 集群间互相调用

HTTP：网络传输协议 基于TCP/IP 规定数据传输格式 缺点是消息封装比较臃肿，传输速度慢

优点是对服务提供和调用方法没有任何技术限定，自由灵活，更符合微服务理念。

常见Http客户端工具：HttpClient、OKHttp、URLConnection。

组件1：注册中心 Eureka Zookeeper

Eureka：服务的管理 注册和发现 状态监管 动态路由

- 1、负责管理记录服务提供者的信息 服务调用者无需自己寻找服务 Eureka自动匹配服务给调用者
- 2、Eureka与服务之间通过心跳机制进行监控

Eureka 就是服务注册中心，对外暴露自己的地址

服务提供者：启动后向Eureka注册自己的信息（地址）

服务消费者：向Eureka订阅服务，Eureka会将对应服务的所有提供者地址列表发送给消费者，并且定期更新

心跳（续约）：提供者定期通过http方式向Eureka刷新自己的状态。

使用：

1、添加依赖 eureka-server依赖

2、application.yml配置

配置端口号 url地址 应用名称

3、启动类上加@EnableEurekaServer 注解

```

server:
  port: 7001    #端口号
spring:
  application:
    name: eureka-server # 应用名称, 会在Eureka中作为服务的id标识 (serviceId)
eureka:
  client:
    register-with-eureka: false    #是否将自己注册到Eureka中
    fetch-registry: false    #是否从eureka中获取服务信息
    service-url:
      defaultZone: http://localhost:7001/eureka # EurekaServer的地址

```

服务提供者:

- 1.引入eureka客户端依赖包 eureka-client
- 2.在application.yml中配置Eureka服务地址
- 3.在启动类上添加@EnableDiscoveryClient或者@EnableEurekaClient (该注解只用于eureka)

```

server:
  port: 18081
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: itcast
    url: jdbc:mysql://127.0.0.1:3306/springcloud?
    useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
  application:
    name: user-provider #服务的名字,不同的应用, 名字不同, 如果是集群, 名字需要相同
#指定eureka服务地址
eureka:
  client:
    service-url:
      # EurekaServer的地址
      defaultZone: http://localhost:7001/eureka

```

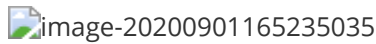
服务消费者: 与提供者基本相同

- 1.引入eureka客户端依赖包 eureka-client依赖
- 2.在application.yml中配置Eureka服务地址
- 3.在启动类上添加@EnableDiscoveryClient或者@EnableEurekaClient

```

server:
  port: 18082
spring:
  application:
    name: user-consumer    #服务名字
#指定eureka服务地址
eureka:
  client:
    service-url:
      # EurekaServer的地址
      defaultZone: http://localhost:7001/eureka

```



服务消费者通过Eureka访问服务提供者

- 1、注入DiscoveryClient对象 用来发现注册中心的服务对象
- 2、获取指定的实例 discoveryClient.getInstances (服务名) 是个list集合 可能有集群 取出第一个 get (0)
- 3、拼接服务地址 http: // + gethost+getport+路由uri;
restTemplate.getForObject (url, 返回值类型。class)

使用IP访问配置

上面的请求地址是服务状态名字，其实也是当前主机的名字，可以通过配置文件，将它换成IP，修改application.yml配置文件，

```
instance:
  #指定IP地址
  ip-address: 127.0.0.1
  #访问服务的时候，推荐使用IP
  prefer-ip-address: true
```

服务续约 心跳检测

服务注册完成以后，**服务提供者**会维持一个心跳，保存服务处于存在状态。这个称之为服务续约 (renew)。



1. 两个参数可以修改服务续约行为
lease-renewal-interval-seconds: 90, 租约到期时效时间，默认90秒
lease-expiration-duration-in-seconds: 30, 租约续约间隔时间，默认30秒
2. 服务超过90秒没有发生心跳，EurekaServer会将服务从列表移除 [前提是EurekaServer关闭了自我保护]

服务消费者 会每隔30s去注册中心重新获取并更新数据，缓存到本地 

```
registry-fetch-interval-seconds: 30
```

失效剔除

服务中心每隔一段时间(默认60秒)将清单中没有续约的服务剔除。
通过eviction-interval-timer-in-ms配置可以对其进行修改，单位是毫秒




AP特性


从CAP理论看，Eureka是一个AP系统，优先保证可用性A 和 分区容错性P，不保证强一致性C，但由于 架构了使用了较多缓存 保证了最终一致性

自我保护：

Eureka会统计服务实例最近15分钟心跳续约的比例是否低于85%，如果低于则会触发自我保护机制。

 image-20200901171855616

Eureka的注册表 registry

 image-20200901210332657

注册表的数据结构： ConcurrentHashMap

ConcurrentHashMap 的 **key 就是 服务名称** **value 是一个服务的多个服务实例 是一个Map集合**；
Map< String , Lease> 这个map的key是服务实例的id

value值 InstanceInfo 代表服务实例的具体信息，比如机器的ip地址、hostname以及端口号

Lease 里面维护了每个服务最近一次发送心跳的时间；

Eureka Server

在高可用架构中，Eureka Server也可以注册到其他server上去，多节点相互注册组成集群。集群间互相视为peer。


Eureka Client向Server注册、续约、更新状态时，接受节点 也就是server更新自己的服务注册信息后，会（逐步）同步到其他peer节点；

但如果是单向注册，比如A注册到B上，A接受数据会同步到B 但B不是A的peer B接受数据不会同步到A；

缓存机制

Server 中存在三个变量保存服务信息。

registry、readWriteCacheMap、readOnlyCacheMap

 image-20200902092002947

eureka里面的三级缓存

 image-20200902092343156

配置项：

1、eureka.server.userReadOnlyResponseCach： true

client直接从readwritecachemap 更新数据 默认为true 设置为false 则可以直接从readWriteCacheMap更新数据

2、从readwrite更新到readonly上的时间周期 默认30s 30000ms

3、清理未续约节点 周期 默认60s 60000ms

4、未续约节点 超时时间 默认90s

Eureka Client

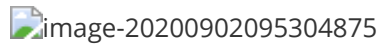
存在两种角色：服务提供者和服务消费者

消费者一般配合Ribbon负载均衡使用；

Client一旦启动，消费者会立即向Server全量更新服务注册信息，之后默认每30s增量更新

提供者会立即向Server注册，默认每30s续约；

(Ribbon 延迟1s向Client获取使用的服务注册信息，默认每30s更新使用的服务注册信息，只保存状态在线的服务)



服务注册中心在选择使用Eureka时说明已经接受了其优先保证可用性(A)和分区容错性(P)、不保证强一致性(C)的特点。如果需要优先保证强一致性(C)，则应该考虑使用ZooKeeper等CP系统作为服务注册中心。

组件2：网关 GateWay

Gateway 主要功能： 1、提供了统一的路由方式 2、（基于过滤链）提供了网关的基本功能：安全、监控和限流

本身也是一个微服务 需要注册到Eureka

核心功能：过滤和路由

核心概念：路由 断言函数 过滤器

application.yml 配置

```
spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          #uri: http://localhost:18081
          #lb协议表示从Eureka注册中心获取服务请求地址
          #user-provider访问的服务名称。
          #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
          uri: lb://user-provider
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/user/**
```

路由地址一般写服务提供者的名称 加上lb://协议 会自动使用负载均衡访问对应服务 表示从Eureka注册中心访问的服务名称

过滤器

过滤器作为Gateway的重要功能。常用于请求鉴权、服务调用时长统计、修改请求或响应header、限流、去除路径等等...

默认过滤器：出厂自带，实现好了拿来就用，不需要实现

全局默认过滤器

局部默认过滤器

自定义过滤器：根据需求自己实现，实现后需配置，然后才能用哦。


全局过滤器：作用在所有路由上。

局部过滤器：配置在具体路由下，只作用在当前路由上。

全局过滤器：对输出响应头设置属性

对输出的响应设置其头部属性名称为X-Response-Default-MyName,值为itheima

image-20200902143343276

image-20200902141743505

局部过滤器：通过局部默认过滤器，修改请求路径。局部过滤器在这里介绍两种：添加路径前缀、去除路径前缀。

在gateway中可以通过配置路由的过滤器PrefixPath 实现映射路径中的前缀

在gateway中通过配置路由过滤器StripPrefix，实现映射路径中地址的去除。通过StripPrefix=1来指定路由要去掉的前缀个数。如：路径/api/user/1将会被路由到/user/1。

配置：

```
spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          #uri: http://localhost:18081
          #lb协议表示从Eureka注册中心获取服务请求地址
          #user-provider访问的服务名称。
          #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
          uri: lb://user-provider
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/**
          filters:
            # 请求地址添加路径前缀过滤器
            - PrefixPath=/user
      default-filters:
        - AddResponseHeader=X-Response-Default-MyName,itheima
```

自定义过滤器

自定义过滤器也有两个：全局自定义过滤器，和局部自定义过滤器。

自定义全局过滤器的案例，自定义局部过滤器的案例。

自定义全局过滤器的案例：模拟登陆校验。

基本逻辑：如果请求中有Token参数，则认为请求有效放行，如果没有则拦截提示授权无效

实现步骤：

- 1、实现接口 GlobalFilter,和Ordered接口 重写GlobalFilterfilter方法 里面获取请求参数 判断是否存在
重写Ordered接口中的 getOrder () 方法 返回值越小 越靠前越早执行

```
@Component
public class LoginGlobalFilter implements GlobalFilter, Ordered {

    /**
     * 过滤拦截
     * @param exchange
     * @param chain
     * @return
     */
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        //获取请求参数
        String token = exchange.getRequest().getQueryParams().getFirst("token");

        //如果token为空，则表示没有登录
        if(StringUtils.isEmpty(token)){
            //没登录,状态设置403
            exchange.getResponse().setStatusCode(HttpStatus.PAYLOAD_TOO_LARGE);
            //结束请求
            return exchange.getResponse().setComplete();
        }

        //放行
        return chain.filter(exchange);
    }

    /**
     * 定义过滤器执行顺序
     * 返回值越小，越靠前执行
     * @return
     */
    @Override
    public int getOrder() {
        return 0;
    }
}
```

3、组件3：负载均衡Ribbon

Ribbon是Netflix发布的负载均衡器，有助于控制HTTP客户端行为。为Ribbon配置服务提供者地址列表后，Ribbon就可基于负载均衡算法，自动帮助服务消费者请求。

Ribbon默认提供的负载均衡算法：**轮询，随机,重试法,加权**。当然，我们可用自己定义负载均衡算法

开启负载均衡

(1)客户端开启负载均衡

Eureka已经集成Ribbon，所以无需引入依赖,要想使用Ribbon，直接在RestTemplate的配置方法上添加@LoadBalanced注解即可

(2)采用服务名访问配置

不再手动获取ip和端口，而是直接通过服务名称调用，

负载均衡策略

1、轮询 默认的策略

2、随机

3、重试法 先按照轮询获取服务 若获取服务失败 则在指定的时间内进行重试，获取可用的服务

4、加权 会根据平均响应时间计算所有服务的权重，响应时间越快 服务权重越大 被选中的概率越大。刚启动时如果统计信息不足，则使用轮询的策略，等统计信息足够会切换到自身规则

```
user-provider:
  ribbon:
    NFLoadBalancerRuleClassName:
      com.netflix.loadbalancer.ZoneAvoidanceRule
```

负载均衡源码分析

负载均衡器动态从服务中心获取服务提供的访问地址（host port）

其中有个负载均衡器 **LoadBalancerInterceptor** 对RestTemplate的请求进行拦截，然后从Eureka上根据服务器名称（user-provider）获取服务列表，随后利用负载均衡算法得到真正服务地址信息 替换掉服务名称 生成id 访问

1、LoadBalancerInterceptor里面有个interceptor方法，拦截RestTemplate方法，从request中获取url，从url中取出请求的服务名称 继续执行execute ()方法

2、根据服务名 getLoadBalancer (servicedId) 获取负载均衡器 和获取服务器的信息getServer方法 得到ip和端口号 获取到节点

负载均衡的切换:在LoadBalancerInterceptor中获取服务的名字，通过调用RibbonLoadBalancerClient的execute方法，并获取ILoadBalancer负载均衡器，然后**根据ILoadBalancer负载均衡器查询出要使用的节点，再获取节点的信息，并实现调用**

组件4：熔断器 Hystrix

Hystrix是Netflix开源的一个延迟和容错库，用于隔离访问远程服务、第三方库、防止出现**级联失败也就是雪崩效应**。

雪崩效应

1. 微服务中，一个请求可能需要多个微服务接口才能实现，会形成复杂的调用链路。
2. 如果某服务出现异常，请求阻塞，用户得不到响应，容器中线程不会释放，于是越来越多用户请求堆积，越来越多线程阻塞。
3. 单服务器支持线程和并发数有限，请求如果一直阻塞，会导致服务器资源耗尽，从而导致所有其他服务都不可用，从而形成雪崩效应；

Hystrix解决雪崩问题的手段，主要是**服务降级（兜底）**，**线程隔离**；

1. 线程隔离：是指Hystrix为每个依赖服务调用一个小的线程池，如果线程池用尽，调用立即被拒绝，默认不采用排队。
2. 服务降级(兜底方法)：优先保证核心服务，而非核心服务不可用或弱可用。触发Hystrix服务降级的情况：线程池已满、请求超时。

熔断原理：

熔断器状态机有3个状态：

关闭状态： 正常访问 ； 打开状态： 所有请求都会降级 ；

半开状态： open状态不是永久的 打开一会后进入休眠时间（默认5s），休眠时间过后会进入半开状态。

默认失败比例的阈值是50%

1. **closed**: 关闭状态，所有请求正常访问
2. **open**: 打开状态，所有请求都会被降级。
Hystrix会对请求情况计数，当一定时间失败请求百分比达到阈(yu: 四声)值(极限值)，则触发熔断，断路器完全关闭
默认失败比例的阈值是50%，请求次数最低不少于20次
3. **Half open**: 半开状态
open状态不是永久的，打开一会后会进入休眠时间(默认5秒)。休眠时间过后会进入半开状态。
半开状态：熔断器会判断下一次请求的返回状况，如果成功，熔断器切回closed状态。如果失败，熔断器切回open状态。
threshold reached 到达阈(yu: 四声)值
under threshold 阈值以下

熔断策略配置

1. 熔断后休眠时间: sleepWindowInMilliseconds 默认为5s
2. 熔断触发最小请求次数: requestVolumeThreshold 默认值是20
3. 熔断触发错误比例阈值: errorThresholdPercentage 默认值是50%
4. 熔断超时时间: timeoutInMilliseconds 默认值为1s

```
# 配置熔断策略:
hystrix:
  command:
    default:
      circuitBreaker:
        # 强制打开熔断器 默认false关闭的。测试配置是否生效
        forceOpen: false
        # 触发熔断错误比例阈值，默认值50%
        errorThresholdPercentage: 50
        # 熔断后休眠时长，默认值5秒
        sleepWindowInMilliseconds: 10000
```

```
# 熔断触发最小请求次数，默认值是20
requestVolumeThreshold: 10
execution:
  isolation:
    thread:
      # 熔断超时设置，默认为1秒
      timeoutInMilliseconds: 2000
```

使用:

- 1、导入依赖：spring-cloud-starter-netflix-hystrix
- 2、开启熔断的注解：**@EnableCircuitBreaker**
- 3、添加降级调用方法，写返回的业务逻辑
- 4、在需要降级处理调用的方法上加上注解 **@HystrixCommand**

```
@HystrixCommand(fallbackMethod = "降级方法名")
@GetMapping(value =("/{id}")
public User queryById(@PathVariable(value = "id") Integer id){
    String url = "http://user-provider/user/find/"+id;
    return restTemplate.getForObject(url, User.class)
}
```

类上默认服务降级的fallback兜底方法 **@DefaultProperties (defaultFallback="方法名")**

刚才把**fallback**写在了某个业务方法上，如果方法很多，可以将**FallBack**配置加在类上，实现默认**FallBack**
@DefaultProperties(defaultFallback="defaultFailBack")，在类上，指明统一的失败降级方法；

组件5：服务通信 Feign

RestTemplate介绍

- RestTemplate是Rest的HTTP客户端模板工具类
- 对基于Http的客户端进行封装
- 实现对象与JSON的序列化与反序列化
- 不限定客户端类型，目前常用的3种客户端都支持：HttpClient、OKHttp、JDK原生URLConnection(默认方式)

组件6：配置中心 config

Config简介

为了方便配置文件集中管理，需要分布式配置中心组件。在Spring Cloud中，提供了Spring Cloud Config，它支持配置文件放在配置服务的本地，也支持配置文件放在远程仓库Git(GitHub、码云)。配置中心本质上是一个微服务，同样需要注册到Eureka服务中心！

使用:

- 1、添加config依赖 spring cloud config
- 2、创建启动类 3注解 @springbootapplication @enableDiscoveryClient @enableConfigServer
- 3、application.yml 配置文件 git.uri: 配置远程仓库码云

```
# 注释版本
server:
  port: 18085 # 端口号
spring:
  application:
    name: config-server # 应用名
  cloud:
    config:
      server:
        git:
          # 配置gitee的仓库地址
          uri: https://gitee.com/sk1111/config.git
# Eureka服务中心配置
eureka:
  client:
    service-url:
      # 注册Eureka Server集群
      defaultZone: http://127.0.0.1:7001/eureka
# com.itheima 包下的日志级别都为Debug
logging:
  level:
    com: debug
```

存在的问题:

修改码云上的配置后，发现项目中的数据仍然没有变化，只有项目重启后才会变化。

组件7: 消息总线 Bus

简介:

SpringCloudBus基于RabbitMQ实现，默认使用本地的消息队列服务，所以需要安装并启动RabbitMQ

Bus用轻量级的消息代理将分布式的节点连接起来，可以用于广播配置文件的更改或者服务的监控管理。

广播出去的配置文件服务会进行本地缓存

消息总线bug+rabbitmq 可以实现 配置中心的配置自动更新，不需要重启微服务

步骤:

- 1、加入依赖 RabbitMQ 和Bus依赖 +监控 actuator依赖
- 2、配置文件
- 3、添加刷新配置 添加一个@RefreshScope

```
# rabbitmq的配置信息：如下配置的rabbit都是默认值，其实可以完全不配置
rabbitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
# 暴露触发消息总线的地址
management:
  endpoints:
    web:
      exposure:
        # 暴露触发消息总线的地址
        include: bus-refresh
```

网络IO

网络七层架构：

7 层模型主要包括：

1. 物理层：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由 1、0 转化为电流强弱来进行传输,到达目的地后在转化为 1、0，也就是我们常说的模数转换与数模转换）。这一层的数据叫做比特。
2. 数据链路层：主要将从物理层接收的数据进行 MAC 地址（网卡的地址）的封装与解封装。常把这一层的数据叫做帧。在这一层工作的设备是交换机，数据通过交换机来传输。
3. 网络层：主要将从下层接收到的数据进行 IP 地址（例 192.168.0.1)的封装与解封装。在这一层工作的设备是路由器，常把这一层的数据叫做数据包。
4. 传输层：定义了一些传输数据的协议和端口号（WWW 端口 80 等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与 TCP 特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如 QQ 聊天数据就是通过这种方式传输的）。主要是将从下层接收的数据进行分段进行传输，到达目的地后在进行重组。常常把这一层数据叫做段。
5. 会话层：通过传输层（端口号：传输端口与接收端口）建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是 IP 也可以是 MAC 或者是主机名）
6. 表示层：主要是进行对接收的数据进行解释、加密与解密、压缩与解压缩等（也就是把计算机能够识别的东西转换成人能够能识别的东西（如图片、声音等））
7. 应用层 主要是一些终端的应用，比如说FTP（各种文件下载），WEB（IE浏览），QQ之类的（你就把它理解成我们在电脑屏幕上可以看到的東西。就是终端应用）。

数据包说明

源端口号（16 位）：它（连同源主机 IP 地址）标识源主机的一个应用进程。

2. 目的端口号（16 位）：它（连同目的主机 IP 地址）标识目的主机的一个应用进程。这两个值加上 IP 报头中的源主机 IP 地址和目的主机 IP 地址唯一确定一个 TCP 连接。
3. 顺序号 seq（32 位）：用来标识从 TCP 源端向 TCP 目的端发送的数据字节流，它表示在这个报文段中的第一个数据字节的顺序号。如果将字节流看作在两个应用程序间的单向流动，则

TCP 用序号对每个字节进行计数。序号是 32bit 的无符号数，序号到达 $2^{32} - 1$ 后又从 0 开始。当建立一个新的连接时，SYN 标志变 1，序号字段包含由这个主机选择的该连接的初始序号 ISN（Initial Sequence Number）。

4. 确认号 ack（32 位）：包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。**只有 ACK 标志为 1 时确认序号字段才有效。** TCP 为应用层提供全双工服务，这意味着数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

5. 控制位（control flags，6 位）：在 TCP 报头中有 6 个标志比特，它们中的多个可同时被设置为 1。依次为：

URG：为 1 表示紧急指针有效，为 0 则忽略紧急指针值。

ACK：为 1 表示确认号有效，为 0 表示报文中不包含确认信息，忽略确认号字段。

PSH：为 1 表示是带有 PUSH 标志的数据，指示接收方应该尽快将这个报文段交给应用层而不用等待缓冲区装满。

RST：用于复位由于主机崩溃或其他原因而出现错误的连接。它还可以用于拒绝非法的报文段和拒绝连接请求。一般情况下，如果收到一个 RST 为 1 的报文，那么一定发生了某些问题。

SYN：同步序号，为 1 表示连接请求，用于建立连接和使序号同步（synchronize）。

FIN：用于释放连接，为 1 表示发送方已经没有数据发送了，即关闭本方数据流。

三次握手

第一次握手：主机 A 发送位码为 $\text{syn} = 1$ ，随机产生 $\text{seq number} = 1234567$ 的数据包到服务器，主机 B 由 $\text{SYN} = 1$ 知道，A 要求建立联机；

第二次握手：主机 B 收到请求后要确认联机信息，向 A 发送 $\text{ack number} = (\text{主机 A 的 seq} + 1)$ ， $\text{syn} = 1$ ， $\text{ack} = 1$ ，随机产生 $\text{seq} = 7654321$ 的包

第三次握手：主机 A 收到后检查 ack number 是否正确，即第一次发送的 $\text{seq number} + 1$ ，以及位码 ack 是否为 1，若正确，主机 A 会再发送 $\text{ack number} = (\text{主机 B 的 seq} + 1)$ ， $\text{ack} = 1$ ，主机 B 收到后确认 seq 值与 $\text{ack} = 1$ 则连接建立成功。

四次挥手

关闭客户端到服务器的连接：首先客户端 A 发送一个 FIN，用来关闭客户到服务器的数据传送，然后等待服务器的确认。其中终止标志位 $\text{FIN} = 1$ ，序列号 $\text{seq} = u$

2) 服务器收到这个 FIN，它发回一个 ACK，确认号 ack 为收到的序号加 1。

3) 关闭服务器到客户端的连接：也是发送一个 FIN 给客户端。

4) 客户端收到 FIN 后，并发回一个 ACK 报文确认，并将确认序号 seq 设置为收到序号加 1。

首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

IO流

流的本质是对文件的处理；IO流主要分为字符流和字节流；字节流分为 `InputStream` 和 `OutputStream`；字符流分为 `Reader` 和 `Writer`；

NIO

NIO 主要有三大核心部分：Channel（通道），Buffer（缓冲区）、Selector（选择器）；

标准的 IO 基于字节流和字符流进行操作，而 NIO 是基于通道（Channel）和缓存区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中

NIO主要有三大核心部分：**Channel**(通道)，**Buffer**(缓冲区)，**Selector**(选择器)。传统IO基于字节流和字符流进行操作，而NIO基于**Channel**和**Buffer**进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。**Selector**(选择器)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。NIO和传统IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。

NIO 的缓冲区

Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。

NIO 的缓冲导向方法不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。**Buffer**，顾名思义 缓冲区，实际上是一个容器，是一个连续数组。**Channel** 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由**Buffer**。

Channel

首先说一下**Channel**，国内大多翻译成“通道”。**Channel**和IO中的**Stream**(流)是差不多一个等级的。只不过 **Stream**是单向的，譬如：**InputStream**，**OutputStream**，而**Channel**是双向的，既可以用来进行读操作，又可以用来进行写操作。NIO中 **Channel**的主要实现有：

1. **FileChannel**
2. **DatagramChannel**
3. **SocketChannel**
4. **ServerSocketChannel**

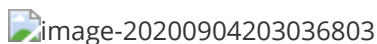
这里看名字就可以猜出个所以然来：分别可以对应文件IO、UDP和TCP（**Server** 和 **Client**）

Selector

Selector类是NIO的核心类，**selector**能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。

NIO和BIO

NIO比传统的BIO核心区别就是，NIO采用的是多路复用的IO模型，普通的IO用的是阻塞的IO模型，两个之间的效率肯定是多路复用效率更高



多路复用：多个客户端注册到Channel通道到多路复用器，一个处理线程轮询Channel，发现哪个满足状态就去处理哪个。

当一个"通道"注册到选择器Selector后，选择器Selector内部就创建一个SelectionKey对象，里面封装了这个通道和这个选择器的映射关系。- 通过SelectionKey的channel()方法，可以获取它内部的通道对象。

NIO和AIO

NIO：同步非阻塞，发出IO请求后，由线程不断尝试获取IO权限，获取到后应用程序自己进行IO操作；

AIO：异步非阻塞，发出IO请求后，由操作系统自己去获取IO权限，再通知服务器应用启动线程处理。

异步非阻塞IO

非阻塞： 服务器(accept)不用等待,可以继续做其他的事情 客户端connect不会等待,可以继续做其他的事情

异步：不用轮询获取监听客户端,有客户端请求服务器,会触发回调函数(CompletionHandler),来处理这个请求

Channel通道

实现同步非阻塞的**服务器**相关的类:

java.nio.channels.ServerSocketChannel:针对面向流的侦听套接字的可选择通道

获取对象的方式: 静态方法打开通道

static ServerSocketChannel **open()** 打开服务器套接字通道

成员方法:

ServerSocketChannel **bind**(SocketAddress local) **给服务器绑定端口号** SocketChannel **accept()**

监听客户端的请求

设置服务器的非阻塞模式

SelectableChannel **configureBlocking**(boolean block) **设置阻塞模式 true:阻塞 false:非阻塞**

客户端: **SocketChannel.open()**; 获取SocketChannel对象

socketChannel.configureBlocking(true); 默认true:阻塞

socketChannel.configureBlocking(false); false:非阻塞

boolean connect(SocketAddress remote) 根据服务器的ip地址和端口号连接服务器

客户端设置为**阻塞模式**:connect方法会多次尝试连接服务器

连接服务器成功connect方法返回true

连接服务器失败,会抛出连接异常

客户端设置为**非阻塞模式**:connect方法只会连接一次服务器

connect方法无论连接成功还是失败都返回false

所以客户端设置为非阻塞模式没有意义

Selector 选择器

如果不使用“多路复用”，服务器端需要开很多线程处理每个端口的请求。如果在高并发环境下，造成系统性能下降。

多路复用器中保存Channel对应的selectkey 调用完删除selectkey

异步非阻塞服务器AIO

创建AIO的服务器端：`java.nio.channels.AsynchronousServerSocketChannel`：用于面向流的侦听套接字的异步通道。

获取对象的方法：

`static AsynchronousServerSocketChannel open()` 打开异步服务器套接字通道。 成员方法：

`AsynchronousServerSocketChannel bind(SocketAddress local)` 给服务器绑定端口号

`void accept(A attachment, CompletionHandler<AsynchronousSocketChannel,? super A> handler)` 接受连接

参数：

`A attachment`:附件,可以传递null

`CompletionHandler<?> handler`:事件处理的类,用于处理accept方法监听到的事件

`CompletionHandler`也叫回调函数,客户端请求服务器之后,会自动执行 `CompletionHandler`接口中的方法 `java.nio.channels.CompletionHandler<V,A>`接口:用于消除异步I / O操作结果 的处理程序。

`void completed•(V result, A attachment)` 客户端连接服务器成功执行 的方法

`void failed•(Throwable exc, A attachment)` 客户端连接服务器失败执行 的方法

步骤：

- 1、创建异步非阻塞服务器对象`AsynchronousServerSocketChannel`对象
- 2、bind方法绑定
- 3、accept方法监听客户端的请求

补充一下：NIO和Netty的工作模型对比？

(1) NIO的工作流程步骤：

1. 首先是先创建`ServerSocketChannel` 对象，和真正处理业务的线程池
2. 然后给刚刚创建的`ServerSocketChannel` 对象进行绑定一个对应的端口，然后设置为非阻塞
3. 然后创建`Selector`对象并打开，然后把这`Selector`对象注册到`ServerSocketChannel` 中，并设置好监听的事件，监听 `SelectionKey.OP_ACCEPT`
4. 接着就是`Selector`对象进行死循环监听每一个`Channel`通道的事件，循环执行 `Selector.select()` 方法，轮询就绪的 `Channel`
5. 从`Selector`中获取所有的`SelectorKey`（这个就可以看成是不同的事件），如果`SelectorKey`是处于 `OP_ACCEPT` 状态，说明是新的客户端接入，调用 `ServerSocketChannel.accept` 接收新的客户端。
6. 然后对这个把这个接受的新客户端的`Channel`通道注册到`ServerSocketChannel`上，并且把之前的 `OP_ACCEPT` 状态改为`SelectionKey.OP_READ`读取事件状态，并且设置为非阻塞的，然后把当前的这个`SelectorKey`给移除掉，说明这个事件完成了
7. 如果第5步的时候过来的事件不是`OP_ACCEPT` 状态，那就是`OP_READ`读取数据的事件状态，然后调用本文章的上面的那个读取数据的机制就可以了

(2) Netty的工作流程步骤：

1. 创建 NIO 线程组 `EventLoopGroup` 和 `ServerBootstrap`。
2. 设置 `ServerBootstrap` 的属性：线程组、`SO_BACKLOG` 选项，设置 `NioServerSocketChannel` 为 `Channel`，设置业务处理 `Handler`
3. 绑定端口，启动服务器程序。
4. 在业务处理 `TimeServerHandler` 中，读取客户端发送的数据，并给出响应

(3) 两者之间的区别：

1. OP_ACCEPT 的处理被简化，因为对于 accept 操作的处理在不同业务上都是一致的。
2. 在 NIO 中需要自己构建 ByteBuffer 从 Channel 中读取数据，而 Netty 中数据是直接读取完成存放在 ByteBuf 中的。相当于省略了用户进程从内核中复制数据的过程。
3. 在 Netty 中，我们看到有使用一个解码器 FixedLengthFrameDecoder，可以用于处理定长消息的问题，能够解决 TCP 粘包读半包问题，十分方便。

Netty 是一款基于 NIO (Nonblocking IO, 非阻塞IO) 开发的网络通信框架，内部有两个线程池，boss 和work线程池 boss线程池负责接收事件的请求，吧对应socket的请求封装到一个NioSocketChannel 中，并交给work线程池负责其read和write请求，由对应的handler处理。

TCP粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象

Netty使用解码器解决这个问题

TCP/IP

TIP/IP协议是指整个TCP/IP协议族；

主要可以分四层：应用层 传输层 网络层 网络接口层

网络访问层：主机必须使用某种协议与网络相连

网络层：功能是使主机可以分组发往任何网络，互联网使用ip协议（互联网协议）

传输层：在这一层定义两个端到端的协议（TCP/UDP）；

应用层：包含所有的高层协议 如域名服务DNS,超文本传送协议HTTP

数据包

数据包是TCP/ip协议通信传输中的数据单位，是将数据打包进行传输，里面包含了 源端口号（16位），目的端口号（16位），顺序号seq（32位）、确认号ack（32位）、控制位（6位 ACK/SYN/FIN等）

TCP和UDP的区别

TCP：一种面向连接的传输层协议,提供可靠的报文传输

UDP：面向无连接的不可靠协议

TCP连接需要三次握手和四次挥手，会有延时，开销大，实时性差；UDP无需建立连接，实时性好，开销小但有丢包；

传输流程

1：地址解析

如用客户端浏览器请求这个页面：<http://localhost.com:8080/index.htm> 从中分解出协议名、主机名、端口、对象路径等部分，对于我们的这个地址，解析得到的结果如下：

协议名：http

主机名：localhost.com

端口：8080

对象路径：/index.htm

在这一步，需要域名系统 DNS 解析域名 localhost.com,得主机的 IP 地址。

2：封装 HTTP 请求数据包

把以上部分结合本机自己的信息，封装成一个 HTTP 请求数据包

3：封装成 TCP 包并建立连接

封装成 TCP 包，建立 TCP 连接（TCP 的三次握手）

4：客户机发送请求命令：

建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是 MIME 信息包括请求修饰符、客户机信息和可内容。

5：服务器响应

服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括服务器信息、实体信息和可能的内容。

6：服务器关闭 TCP 连接

服务器关闭 TCP 连接：一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接，然后如果浏览器或者服务器在其头信息加入了这行代码 **Connection:keep-alive**，TCP 连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

对于TCP连接：

1.服务器端1) 创建套接字create; 2) 绑定端口号bind; 3) 监听连接listen; 4) 接受连接请求accept, 并返回新的套接字; 5) 用新返回的套接字recv/send; 6) 关闭套接字。

2.客户端1) 创建套接字create; 2) 发起建立连接请求connect; 3) 发送/接收数据send/recv; 4) 关闭套接字。

TCP总结：

Server端：create – bind – listen-- accept-- recv/send-- close

Client端：create----- connct-----send/recv-----close.

软件结构

C/S结构：全称为Client/Server结构，是指客户端和服务端结构。常见程序有QQ、迅雷等软件。

B/S结构：全称为Browser/Server结构，是指浏览器和服务端结构。常见浏览器有谷歌、火狐等。

Socket通信

表示TCP通信中的客户端的类

java.net.socket：此类实现套接字，套接字是两台机器间通信的端点。

套接字:封装了IP地址和端口号的网络单位

主要方法：获取此套接字的输入流和输出流。

OutputStream getOutputStream()返回此套接字的输出流。

InputStream getInputStream() 返回此套接字的输入流。

客户端和服务端之间进行数据交互,不能使用自己创建的流对象
必须使用Socket对象中提供的流

表示服务器的类:

java.net.ServerSocket:此类实现服务器套接字。

成员方法:

Socket accept() 侦听并接受到此套接字的连接。

客户端请求服务器,服务器就必须明确是哪个客户端请求的服务器

使用accpet方法,一直监听客户端的请求,有客户端请求服务器,accpet方法就会获取到请求的客户端Socket对象

Dubbo

1. 服务容器 负责启动, 加载, 运行服务提供者。
2. 服务提供者 在启动时, 向注册中心注册自己提供的服务。
3. 服务消费者 在启动时, 向注册中心订阅自己所需的服务。
4. 注册中心 返回服务提供者地址列表给消费者, 如果有变更, 注册中心将基于长连接【推送】变更数据给消费者。
5. 服务消费者, 从提供者地址列表中, 基于软负载均衡算法, 选一台提供者进行调用, 如果调用失败, 再选另一台调用。
6. 服务消费者和提供者, 在内存中累计调用次数和调用时间, 定时每分钟发送一次统计数据到 监控中心Monitor。
- 7.

Zookeeper

1、什么是zookeeper?

zookeeper本质是一个分布式的小文件存储系统, 以目录树的方式存储数据, 并对树中的节点进行有效管理。维护和监控存储的数据的状态变化。

Zookeeper适用于存储和协同相关的关键数据, 不适合用于存储大数据量存储。

分布式系统的描述总结是:

多台计算机构成 计算机之间通过网络通信

彼此进行交互 共同目标

zookeeper采用的是推拉结合的方式

推: 服务端会推给注册了监控节点的客户端Watcher事件通知;

拉: 客户端获得通知后, 然后主动到服务端拉取最新的数据

ZooKeeper 的数据模型是层次模型, 成为data tree。层次模型常见于文件系统。

2、角色

角色:leader、follower、observer

1.Leader作为整个ZooKeeper集群的主节点, 负责响应所有对ZooKeeper状态变更的请求。它会将每个状态更新请求进行排序和编号, 以便保证整个集群内部消息处理的FIFO。

2.Follower的逻辑就比较简单了。除了响应本服务器上的读请求外, follower还要处理leader的提议, 并在leader提交该提议时在本地也进行提交。

如果ZooKeeper集群的读取负载很高, 或者客户端多到跨机房, 可以设置一些observer服务器, 以提高读取的吞吐量。3.Observer和Follower比较相似, 只有一些小区别: 首先observer不属于法定人数, 即不参加选举也不响应提议; 其次是observer不需要将事务持久化到磁盘, 一旦observer被重启, 需要从leader重新同步整个名字空间。

特性:

1. Zookeeper: 一个leader, 多个follower组成的集群
2. 全局数据一致: 每个server保存一份相同的数据副本, client无论连接到哪个server, 数据都是一致的
3. 分布式读写, 更新请求转发, 由leader实施
4. 更新请求顺序进行, 来自同一个client的更新请求按其发送顺序依次执行
5. 数据更新原子性, 一次数据更新要么成功, 要么失败
6. 实时性, 在一定时间范围内, client能读到最新数据

半数存活机制: 只要有半数以上的节点存活, 集群就能提供服务(==适合装在奇数台机器上==)

常用命令:

-e: 表示普通临时节点 -s: 表示带序号节点

1. 查询所有命令 `help`
2. 查询根路径下的节点 `ls /zookeeper`
3. 创建普通永久节点 `create /app1 "helloworld"`
4. 创建带序号永久节点 `create -s /hello "helloworld"`
5. 创建普通临时节点 `create -e /app3 'app3'`
6. 创建带序号临时节点 `create -e -s /app4 'app4'`
7. 查询节点数据 `get /app1`
8. 修改节点数据 `set /app1 'hello'`
9. 删除节点 `delete /hello0000000006`
10. 递归删除节点 `rmr /hello`
11. 查看节点状态 `stat /zookeeper`

MySQL

一条SQL语句的执行过程

首先客户端连接mysql服务器, 连接后执行sql语句, 过程中需要先经过分析器得出它是什么操作, 比如update操作, 接着经过优化器, 决定使用id这个索引, 然后执行器找到这一行数据 再进行更新操作。

redo log 是InnoDB引擎特有, 它属于物理日志, 主要记录某个数据页上做了什么修改, 而且记录空间是固定且会用完的;

bin log 是属于server层特有的, 主要是在执行器中记录日志, bin log 是属于逻辑日志, 它有statement 和 row 两种模式, statement记录的是执行的sql语句, row记录的是更新行的内容, 所以是记录两条, 一条是更新前的内容, 另外一条是更新后的内容。默认模式是 row 模式。另外 bin log 是会追加写入日志, 当日志文件写到一定大小的时候, 就会切换到下一个继续写入日志, 并且不会覆盖之前的日志文件。

MySQL 内部逻辑 1、客户端/服务端通信协议 (jdbc协议) 2、查询缓存 有则放回 3、解析 (sql句法) 4.查询优化器 (决定索引查询)

5、执行引擎执行计划 从存储引擎innoDB中获取数据

索引

- 1、数据库索引方式: B+tree 和Hash;
- 2、索引类型: 四种 普通索引 唯一性索引 主键 全文索引

索引sql语句:

建表时带索引: `CREATE TABLE table_name ([column_name INT ...], INDEX [index_name] column_name);`

修改表添加索引: `ALTER TABLE table_name ADD INDEX [index_name] column_name;`

index_name索引名一般没用到, 只要那一列有了索引, 查询的时候就会自动使用索引。

1、普通索引: 就是最基本类型的索引, 没有唯一性。上面的例子就是普通索引。

2、唯一性索引: 和“普通索引”的区别就是, 是唯一性的。语法: `UNIQUE [index_name] column_name`

3、主键: 不仅唯一, 还只能有一列。(对比上面两个记忆) 语法: `PRIMARY KEY [index_name] column_name`

4、全文索引: 可以作用在varchar和text。关键字为FULLTEXT。用的较少

hash表查询时间比B+tree查询快 为啥还是选择B+树作为索引呢?

和业务场景有关。如果只是查询一个数据, hash查询更快; 但数据库中查询经常查询多条, 这时候由于B+树索引有序, 并且又有链表相连, 所以查询效率比hash表要快的;

另外 数据库的索引一般都是在磁盘上, 数据量大的时候无法一次装入内存 B+树的设计可以允许数据分批加载, (B树需要做局部的中部遍历, 可能需要跨层访问, 而 B+ 树由于所有数据都在叶子结点, 不用跨层, 同时由于有链表结构, 只需要找到首尾, 通过链表就能把所有数据取出来了) 树的高度较低, 查询速度快

在内存中 红黑树比B树效率更高但设计到磁盘操作 B树更优

mysql主从复制

数据更新--写binlog日志--发送到从服务器

从服务器开启io线程, 读binlog写入到自己的relay-log日志中, 根据日志内容执行sql线程完成数据更新。

Mybatis

一级缓存 第一次查询会存入缓存中 如果当前会话执行了修改类操作 会清空当前一级缓存

二级缓存 跨session缓存 一个接口内 sqlsession会话对象查询操作都会缓存

常用的键必须有索引 索引的目的是为了查询 频繁查询的需要索引 频繁修改的不要做索引

- 1、主键必须有索引
- 2、where从句 group by 从句 order by 从句 on 从句后面的索引添加
- 3、复合索引建立

不适合索引:

- 1、经常需要修改的表 比如插入 删除 不要建立太多的索引
- 2、表记录比较少 没必要用索引 (500) 删除无用的所用

如果数据表频繁插入删除 就可以索引重建

mysql查询只会用一个索引;

索引失效: 1、not in != 反向的都会失效

- 2、索引中含有null
- 3、字符串不加单引号 索引失效
- 4、尽量避免子查询
- 5、where子句中对字段进行null值判断
- 6、少用or

MySQL语句优化

- 1、索引中含有null值
- 2、where条件是or 用in来代替
- 3、Not in 也不会走索引
- 4、尽量避免用子查询
- 5、mysql不支持函数索引 因此也会失效
- 6、limit 分页查询 最好写成 取前一页的最大行数id 比如 limit 2232,10; 改成 where id >2231,10;
- 7、group by xx order by null 禁止排序
- 8批量插入合并写语句

mysql常用配置参数：

- 1、基本配置：数据目录：datadir 存放mysql的数据库文件和日志
默认字符集：default-character-set=utf-8 skip-grant-tables 跳过权限表
- 2、日志相关：错误日志：log-error 记录数据更改的查询语句 log-bin
sync-binlog 指定多少次写日志后同步磁盘
- 3、存储引擎：default-table-type： 设置mysql的默认存储引擎
innodb_data_home_dir 指定共享表空间数据文件的路径和大小

Redis

1、redis的序列化机制

- 1、默认的情况下redisTemplate操作key value的时候 必须要求 key一定实现序列化 value 也需要实现序列化
- 2、默认的情况下redisTemplate使用JDK自带的序列化机制：JdkSerializationRedisSerializer
- 3、JDK自带的序列化机制中要求需要key 和value 都需要实现Serializable接口
- 4、RedisTemplate支持默认以下几种序列化机制：机制都实现了RedisSerializer接口
 - + OxmSerializer
 - + GenericJackson2JsonRedisSerializer
 - + GenericToStringSerializer
 - + StringRedisSerializer
 - + JdkSerializationRedisSerializer
 - + Jackson2JsonRedisSerializer

我们可以进行自定义序列化机制：例如：我们定义key 为字符串序列化机制，value：为JDK自带的方式，应当处理如下：

```
@SpringBootApplication
@GetterScan(basePackages = "com.itheima.dao")
//MapperScan 用于扫描指定包下的所有的接口，将接口产生代理对象交给spring容器
public class MybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisApplication.class,args);
    }

    @Bean
    public RedisTemplate<Object, Object> redisTemplate(
        RedisConnectionFactory redisConnectionFactory) throws
        UnknownHostException {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        //设置key的值为字符串序列化方式 那么在使用过程中key 一定只能是字符串
        template.setKeySerializer(new StringRedisSerializer());
        //设置value的序列化机制为JDK自带的方式
        template.setValueSerializer(new JdkSerializationRedisSerializer());
        return template;
    }
}
```


redis的基本使用类型

string hash list set zset 项目中主要用的就是redis的string 存储工艺参数

redis的常见问题、redis的持久化机制

缓存击穿（热点数据不过期 或者加锁）

缓存穿透（数据库也没有 业务层可以对查询条件进行判断 如果id小于0 或者过大 直接拒绝 查询不到 返回一个临时null 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力）

缓存雪崩 设置随机过期时间

redis的缓存淘汰策略

- 1、noeviction 默认 写会报错 读可以继续进行
- 2、ttl过时策略 从过期时间的键中删除马上要过期
- 3、lru- volatile 过期时间中找 最近近期使用
- 4、lru- allkeys 所有键 最近近期使用
- 5.6 lfu- volatile allkeys 过期键 所有键 频率中最少的键
- 7.8 random 过期键 所有键 随机删除

redis过期删除策略

惰性删除（查询的时候删除） 定时（cpu） 定期 一定时间

redis是单线程的数据库 我们删除过期键不可能占据大量时间 应该redis的工作；所以定期存在局限性

定时扫描策略：

redis默认每秒扫描10次 从过期键里面随机调20key进行删除。如果删除完 发现过期键的数量高于1/4 说明过期键占用内存还很严重 就继续删除20个 但肯定不能一直删除下去 所以加了一个时间上限 默认不超过25ms。

redis的集群

RabbitMQ

1.MQ消息队列是应用程序之间的通信方法;

任务异步处理:将不需要同步处理的并且耗时长操作由消息队列通知消息接收方进行异步处理;提高了应用程序的响应时间;

应用程序解耦合:MQ相当于一个中介,生产方通过MQ与消费方交互,它将应用程序进行解耦合;流量消峰;

2.实现MQ两个协议:AMQP,JMS;

AMQP:高级消息队列协议,是一个进程间传递异步消息的网络协议,直接定义网络交换的数据格式;

两者区别:JMS定义了统一的接口,对消息操作进行统一;AMQP通过规定协议来统一数据交互的格式;

JMS限定了必须使用java语言;AMQP只是协议,不规定实现方式,跨语言;

JMS只有(订阅模式/点对点消息模式);

3.Kafka,RocketMQ,RabbitMQ;

Kafka:快速持久化,高吞吐,10w/s,高堆积;支持hadoop数据并行加载; 50w

RocketMQ:(**)保证严格的消息顺序;丰富的消息拉取模式;理论上不丢失消息;支持事务消息;亿级消息堆积能力;毫秒 延迟; 10w

RabbitMQ:使用erlang语言编写; 微妙 延迟; 吞吐量 5w

3.1消息队列模式

(1)工作模式:多个消费端共同消费同一个队列中的消息;[3个角色 生产者/消费者/队列]

(2)订阅模式:多了交换机角色,生产者将消息发送给交换机,交换机负责转发;交换机不具备存储消息的能力;要有队列跟其进行绑定;(Fanout广播;Direct定向;Topic通配符); 交换机将消息交给每一个绑定的队列;

(3)路由模式:

1. 队列与交换机的绑定, 不能是任意绑定了, 而是要指定一个RoutingKey (路由key)
2. 消息的发送方在 向 Exchange发送消息时, 也必须指定消息的 RoutingKey。
3. Exchange不再把消息交给每一个绑定的队列, 而是根据消息的Routing Key进行判断, 只有队列的Routingkey与消息的 Routing key完全一致, 才会接收到消息

(4)通配符模式:都是根据routingkey将消息路由到不同的队列;只是以一大类(以**开头/结尾)来区别;

4.RabbitMQ的使用步骤:

(1)导入springboot-amqp整合起步依赖;(2)配置RabbitMQ信息,以及路由交换机,队列和路由规则;在合适的地方发送消息给指定的队列名称(rabbitTemplate.convertAndSend);然后再合适的地方编写监听器接收消息;(@RabbitListener监听队列;@RabbitHandler处理监听事务);

5.开发遇到的问题

高版本的rabbitmq配合低版本的erlang, 会出现资源占用后不被正常释放的问题;

6.生产者可靠性消息投递

MQ投递消息流程:(1)生产者发送消息给交换机;(2)交换机根据routingkey转发消息给队列;(3)消费者监控队列,获取队列中信息;(4)消费成功,删除队列中消息;

(1)提供了两种方式用来控制消息的投递可靠性; confirm模式:生产者发送消息到交换机的时机 return模式:交换机转发消息给queue的时机;

(2)消息从product到exchange会返回一个confirmCallback;消息从exchange到queue投递失败会返回一个returncallback;

7.消费者确认机制(ACK)

(1)三种方式:自动确认;手动确认;根据异常情况确认;

(2)自动确认方式:当消息一旦被Consumer接收到,则自动确认收到,并将相应 message 从 RabbitMQ 的消息缓存中移除。但是在实际业务处理中,很可能消息接收到,业务处理出现异常,那么该消息就会丢失;

(3)手动确认方式:需要在业务处理成功后,调用channel.basicAck(),手动签收,如果出现异常,则调用channel.basicNack()等方法,让其按照业务功能进行处理,比如:重新发送,比如拒绝签收进入死信队列等等;

8.消费端限流

(1)在并发量大的情况下,生产方不停发送消息,消息会在队列中堆积很多,当消费端启动时,会瞬间涌入,那就有可能宕掉;所以采用在消费端进行限流操作,每秒钟放行多个消息;这样就可以进行并发量控制,减轻系统的负载,提供系统的可用性;

9.TTL(Time to live 消息过期时间)

当消息到达存活时间后,还没有被消费,会自动清除;

(1)针对某一队列;如果过期,就清除队列中的全部;(2)针对特定;队列中消息过期,则清除特定消息(**注意**:针对某一个特定的消息设置过期时间时,一定是消息在队列中在队头的时候进行计算,如果某一个消息A 设置过期时间5秒,消息B在队头,消息B没有设置过期时间, B此时过了已经5秒钟了还没被消费。注意,此时A消息并不会被删除,因为它并没有再队头);

10.死信队列

(1)当消息成为死信后,可以被重新发送到另一个交换机(Dead Letter Exchange),死信交换机DLX;

(2)成为死信的三种条件:队列消息长度到达限制(最大消息长度); 消费者拒接消费消息; 原队列存在消息过期设置,消息过期未被消费;

(3)死信交换机,可在任何队列上被指定,实际上就是设置某个队列的属性; 当队列中有死信时,rabbitmq会自动将这个消息重新路由到另一个队列中;

设置参数:x-dead-letter-exchange和x-dead-letter-routing-key;

11.延迟队列

当消息进入队列后不会立即被消费,只有到达指定时间后,才会被消费; 可以根据TTL+死信队列的方式进行达到延迟的效果; 比如:订单业务:用户下单30分钟后未支付,则取消订单;

12.幂等性问题(重复消费)

1.多次请求某一个资源,对于资源本身应该具有同样的结果;超时//网络抖动问题;

(1)通过版本号或者时间戳解决;CAS (2)数据库锁; (3)[消息表]对消息进行判定,消息是有状态的,接收到消息,消费完立即将消费的状态改掉;如果第一次操作完,那么就改掉状态,等到第二次来的时候,一看到消息状态,就会弃用(在事务中,写一个**存储过程**,业务逻辑判断);

2.发生重复消费:要判断该消息的具体事情;[Select不会/update/insert/delete]

例:update set a=? where id =? 情景:第一次执行和第二次执行的区别;(涉及消息的时序性问题) 假如在第一次执行完后将a=4,来了一个线程,将a改为5,第二次执行后,又将a改为4;就出了顺序问题;

13.rabbitmq集群搭建和通信

14.rabbitmq如何保证数据不丢失

(1)保证生产者不丢消息[回调接口](#)保证rabbitmq不丢消息;(3)保证消费端不丢消息;

(1)保证生产者不丢消息:确切说写消息,可以开启confirm模式,如果消息写入mq中,那么就会回传一个ack消息;如果不能处理,则会回调一个nack接口,通知消息接收失败,可以重试;

(2)保证rabbitmq不丢消息:

1.开启rabbitmq的持久化(持久化queue和message),消息写入后会持久到磁盘,如果rabbitmq挂掉,恢复之后会自动读取之前存储的数据,一般不会丢;

2.如果没有持久化,就挂掉,可能会导致少量数据会丢失;

3.持久化message,再发送消息时将消息设置为持久化,这样rabbitmq就会将消息持久化到磁盘上;

(3)保证消费端不丢消息:依靠ACK机制(消费端收到消息要有应答/响应),关闭rabbitmq的自动确认通知,使用手动通知,等消费者服务执行完毕没有异常在ack;

(4)rabbitmq发送消息有重试机制(断网)和应答模式(消费端);默认情况下,rabbitmq会自动实现重试机制,默认5s重试一次,重试策略可修改:最大重试次数以及重试间隔次数;

(5)rabbitmq:在使用amqp,发送异常,消息重回队列;

(6)当投递失败,路由失败,消费失败时,会导致消息失败;

(6)终极保证消息不丢失办法:发送方与消费端两者都有一个**消息表**,用于记录判定**发送**的消息和**接收**的消息是否一致;判定的话根据一个**监控系统**(查询两张表的数据对比),通过定时任务来实现[每个5s生成一个动态图,便于观察];来通过人工干预(增或减)解决; 对于消息的发送和接收同样要做**日志记录**,防止丢失,搭建日志服务平台(ELK),做日志的实时分析; rocketMQ支持**顺序消息/事务消息**;底层采用queue队列保证顺序;

15.MQ事务消息(分布式事务)

在分布式事务中,(发送方)写消息和发送消息要保证原子性,要

么都成功,要么都失败;

[1]MQ发送方,向MQ服务器发送消息(状态HALF),服务器就会响应接收成功,而这并不代表真正的成功,MQ发送方就会去执行本地事务,然后将执行结果(commit/rollback)提交给服务器; 服务器收到后,在进行判定,commit就投递消息,rollback就删消息不投递;

注意:执行超时,如果执行超时,就会在服务器中呈现**unkown**状态; 对此,MQ服务器有个**超时机制**,如果是**unkown**状态就会进行回调,回查事务状态;**发送方**就会去检查**本地事务状态**;询问次数有**上限**,到达上限后,就删除(**做日志**,进行记录,然后人工干预);

16.处理消息堆积

(1)消息堆积主要是生产者和消费者不平衡导致;

(2)处理思路:提高消费者的消费速率,保证消费者不出现消费故障;

(3)避免消息堆积:a.足够多的内存资源;b.消费者数量足够多;c.避免消费者故障;

(4)解决:修复consumer,使其恢复正常消费;临时新建一个topic,将其队列增加10或20倍;然后编写临时处理分发程序,从旧的topic中快速读取到新的topic中,再启动更多consumer消费临时新的topic的消息;直到堆积的消息处理完成,再还原到正常的机器数量;

ES

JVM

1、JVM内存空间

PC寄存器： 创建程序计数器，就是一个指针，指向方法区中的方法字节码

虚拟机栈：JVM Stack ： 八种基本数据类型、对象引用（引用指针）【只存放局部变量】

本地方法栈：Native Method Stack

Java调用非java代码的接口，此方法用非java语言实现

方法区 Method Area

用于存储虚拟机加载的类信息 class信息

堆 Java Heap

储存对象的实例

方法区和堆都是线程共享的，在JVM启动时创建，在JVM停止时销毁，而Java虚拟机栈、本地方法栈、程序计数器是线程私有的，随线程的创建而创建，随线程的结束而死亡。

GC机制

五个内存区域中，有3个是不需要进行垃圾回收的：本地方法栈、程序计数器、虚拟机栈。因为他们的生命周期是和线程同步的，随着线程的销毁，他们占用的内存会自动释放。所以，只有**方法区和堆区**需要进行垃圾回收，回收的对象就是那些不存在任何引用的对象。

判断是否是垃圾数据

根搜索算法：从一个叫GC Roots的根节点出发，向下搜索，如果一个对象不能达到GC Roots的时候，说明该对象不再被引用，可以被回收。

可作为GC Roots的对象包含以下几种：

- 1.虚拟机栈(栈帧中的本地变量表)中引用的对象。
- 2.方法区中静态属性引用的对象
- 3.方法区中常量引用的对象
- 4.本地方法栈中(Native方法)引用的对象

Minor GC机制 和 Full GC机制

我们说的堆内存里面主要分三块： 分别是新生代 和老生代和持久代；

新生代里面大致分为Eden区和Survivor区，Survivor区又分为大小相同的两部分：FromSpace和ToSpace。新建的对象都是从新生代分配内存，Eden区不足的时候，会把存活的对象转移到Survivor区。当新生代里面**Eden区**满时，就会触发Minor GC（也称作Youn GC）。

老生代或者持久代里面垃圾回收机制称为 Full GC机制（Major GC），速度比Minor GC机制慢10倍以上。当老生代满时会触发Full GC，会同时回收新生代、老生代；当持久代【方法区】满时也会触发Full GC，会导致Class、Method元信息的卸载

GC算法：

新生代主要使用复制算法： 将不需要清除的对象复制到一块区域，清除其他的无用对象；缺点是需要额外的空间和移动

老生代主要使用标记压缩算法： 将不需要清除的对象进行标记，清除未标记的对象，然后把活的对象向空闲空间移动，再更新引用对象的指针。效率高，也解决了内存碎片的问题，缺点是需要移动对象。

Survivor分两块区域，from和to 每次gc后就会交换，

2、线上CPU过高的问题解决

1.查询占用cpu最高的线程id

```
ps -eLo pid,lwp,pcpu | grep 15285 | sort -nk 3
```

2.导出JAVA线程栈信息

命令：kill -3 [PID] 或者 jstack [PID]

3.从栈信息中找到线程数多的几个

命令：sort 文件名 | uniq -c | sort -nk 1

4、分析线程数最多的前几个线程

如果是功能点占用cpu高，那么可以从业务和技术两个方面优化：

4/1 弹性时间：对高使用率的请求，分散到不同时间 比如队列，异步，减少统一时间的请求

4/2 批处理或定时任务

如果是GC进程占用cpu更多，则使用JVM自带的性能监控和故障分析工具VisualVM，查看垃圾回收活动是否过于频繁，如果是GC次数过多，就查询堆存储中对象大小最大的对象，根据占用内存最大的对象去查询代码中接口，分析代码问题，查看sql语句是否可以优化，是否有死循环

类加载机制

一个类加载器要去加载一个class时，

1、全盘负责：该类加载器会去加载改class依赖和应用的所有class；

2、双亲委派：会先去找父类加载去尝试从自己的类路径中加载，只有父类加载器无法加载时 自己再去加载

3、缓存机制：加载过的类class文件会被缓存，当程序中要使用某个class时，类加载器会从缓存区中搜寻该class；没有才会读取该类的二进制数据。这也是为什么我们修改了class 必须重启jvm 程序所做的修改才会生效

类加载过程

.java文件 jdk编译器将其编译成.class文件 类加载系统会去加载这个.class文件（类加载器有三种 启动 扩展 系统）就会把.class文件加载到jvm内存中 主要是方法区内保存好类的基本信息 创建出静态变量 之后jvm的执行引擎会去解析我们的字节码文件 生成底层系统指令 在调用本地接口 运行这些指令 使我们的文件能够在不同的操作系统上执行

类加载系统把字节码文件加载到JVM中，主要分为 加载 验证 准备 解析 初始化；

GC 算法

根可达性 GC-Root finalize() 方法 标记垃圾回收的时候执行一次

新生代进去老年代的条件

1、年龄到了 16次的时候 2、对象太大 3、s区相同年龄大于内存的一半 4、s区满了

Full GC

1、老年代满了 2、永久代满了 3、空间分配担保（在进行ygc的时候 e区的所有对象的大小 大于老年代连续可用空间大小 开启了空间担保 历次回收完s区移动 的数据的平均大小 大于老年代的可用连续空间 直接fullgc）

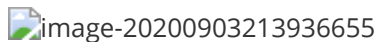
Java对象的引用 强软弱虚

强： new object 直接指定 软： gc结束 如果内存不足的才会回收

弱： gc直接回收 虚：

基础算法

Spring容器启动过程



第一步资源定位，首先我们找到Spring配置文件的位置，创建类ClassPathResource，传入参数配置文件路径path



第二步创建IOC容器类DefaultListableBeanFactory，这个类实现了BeanDefinitionRegistry接口，这个接口有一个回调函数，后面调用能向IOC注册bean的定义信息。

第三步读取配置文件，并将bean注册到IOC容器中去，具体步骤是先创建配置文件读取器XmlBeanDefinitionReader；再调用里面的读取文件的方法 并传入资源类loadBeanDefinitions（resource），从中读出spring的配置信息，并调用Bean注册回调函数向容器中注册Bean；

JDK1.8新特性

1、lambda表达式

Lambda是一个匿名函数,我们可以把lambda表达式理解为是一段可以传递的代码(将代码像数据一样进行传递),写出更简洁,更灵活的代码。

语法： lambda的参数列表 -> lambda表达式需要执行的功能 即lambda体

(参数列表)->{重写抽象方法的方法体}

Lambda表达式作用：简化匿名内部类

lambda表达式使用前提：必须有接口,接口中有且只能有一个抽象方法(函数式接口)

2、函数式接口

Consumer 消费性接口

Supplier 攻击型接口

Function(T,R) 函数向接口

Predicate 断言型接口

3、方法引用和构造器引用

4、Stream API

什么是流？

- 1、流本质其实就是对数据的计算，但流不存储数据值
- 2、流的中心思想是延迟计算 如果把集合作为流的数据源，创建流时不会导致数据流动，如果流的终止操作需要值时，流就会从集合中获取值
- 3、流只能使用一次 使用完得重新创建 比如list.stream();

流使用时一般包括三件事：

- 1、一个数据源（如集合）来执行一个查询；
- 2、一个中间操作链，形成一条流的流水线；
- 3、一个终端操作，执行流水线生成结果；

```
//forEach(),Stream提供的新的方法来迭代流中的每个数据。
list.stream()
    .filter(s->s.startsWith("张"))
    .filter(s->s.length()==3)
    .forEach(System.out::println);

//去重:distinct()
stream.distinct().forEach(System.out::println);

//sorted()排序操作  Comparator<? super T> comparator 传递一个Comparator接口
list.stream().sorted(String::compareTo).forEach(System.out::println);
```

使用流来进行排序

```
List<Student> collect =
list.stream().sorted(Comparator.comparing(Student::getAge).reversed().thenComparing(Comparator.comparing(Student::getName))).collect(Collectors.toList());
```

使用流计算：

串行流：stream();

并行流：paralleastream(); (会有线程安全问题)

5、接口中默认方法与静态方法

6、新时间日期API

Docker

docker命令

1、本地微服务打包 alt+p copy到linux系统

2、创建dockefile 文件 touch文件

#my dockerfile ljh

FROM java:8

MAINTAINER ljh

ADD demo-0.0.1-SNAPSHOT.jar app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

EXPOSE 8080

3、构建镜像

docker build -t +名字

4.别人拉取镜像

dock pull docker

下载镜像

docker pull docker.io/canal/canal-server

容器安装 创建容器

docker run -p 1111: 111 --name canal -d docker.io/canal/canal-server

登录canal

docker exec -it canal /bin/bash

配置 canal

vi canal.properties

配置canal实例 同步配置

vi example/instance.properties

配置完成后并重启canal容器

docker restart canal

启动tomcant tomcat ./startup.sh

Linux的命令

常用命令:

查看磁盘使用空间: `df -hl`复制代码

查看网络是否连通: `netstat`

查看各类环境变量: `env`

查找命令的可执行文件: `whereis [-bfmsu] [-B <目录>...] [-M <目录>...] [-S <目录>...] [文件...]`

`which` 只能查可执行文件 `whereis` 只能查二进制文件、说明文档, 源文件等

当前系统支持的所有命令的列表: `compgen -c`

显示当前所在目录: `pwd`

1、展示目录列表命令 `ls (list)`

`ls` 展示当前可见文件

`ls-a` 所有包括应

`ls -l` 查看某个目录下的文件或目录的权限

`ll` 当前目录所有文件的详细信息

`pwd` 显示当前的目录

2、切换目录命令 `cd`

`cd ..` 切换到上一级

`cd /` 切换到系统根目录下

`cd ~` 切换到当前用户的根目录下

3、文件创建和删除 `touch` 和 `rm`

`touch test.txt`

`rm -f test.txt` 直接删除文件

`rm -rf test` 递归直接删除

4、文件打包和解压

`tar -zcvf +` 打包压缩后的文件名 +要打包压缩的文件

`tar -xvf +文件名` 解压

5、复制和移动

`cp` 复制

`mv` 移动

6、显示

`cat`

7、进程管理

`ps -ef | grep java` 查看当前进程 通过java筛选

`kill -9 pid` 杀死某个进程

8、vim 编辑

9、`wq` 保存 ! `wq` 不保存退出

10、`top`命令 查看cpu

时间问题 统一

1、数据库指定GMT+8

2、

```
jackson:
  data-format: java.text.SimpleDateFormat
  time-zone: GMT+8
```

数据结构：

1、数组： 查询快 容量固定

2、栈： 先入后出

3、队列： 先进先出 初始有两个指针 front和rear 进一个元素 $(rear+1) \% size$ 出一个元素 $front+1$

4、链表： 增删快 每个元素包括两个节点一个储存元素的数据域（内存空间）另一个是指向下一个结点地址的指针域。单链表 双链表 循环链表

双向链表 有两个指针 left 和 right 分别指向左边结点和右边结点。

循环链表： 在单向链表和双向链表的基础上 将两种链表的最后一个结点指向第一个结点

5、散列表：也叫哈希表 类似于hashmap

6、堆： 总是一颗完全二叉树

7、图： 不了解

8、树：

二叉树： 每个结点最多两个子树；左值小于父 右值大于父

完美二叉树： 所有的叶子节点都在同一层，毫无间隙填充了h层；

满二叉树： 每个内部节点（非叶子节点）都包含两个孩子

完全二叉树： 一个高度为h的完美二叉树减少到h-1，并且最底层的槽被毫无间隙地从左到右填充

平衡二叉树： AVL树 左右树都是平衡二叉树，且左右树的高度差的绝对值不超过1。

二叉查找树： 有序二叉树 左子树所有结点的值均小于它的根结点的值。右均大于

红黑树： 平衡排序二叉树的基础上 加了约束 根是黑色 红色节点两个儿子肯定是黑的

二叉树的遍历：

前中后遍历： 后遍历： 从下往上，左右根遍历 中遍历： 从下往上 左根右遍历

前遍历： 从上往下，根左右

m阶B树： 每个结点最多有m个子结点 子节点 $\lfloor m/2 \rfloor, m$ $m/2$ 向下取整 元素中存储关键字数据和指针

每个节点中元素从小到大排列 每个关键字左结点的值 都小于或等于该关键字。右节点的值都大于或等于该关键字。

查询速度并不优于平衡二叉树 但是经过结点少 减少磁盘io的次数。

m阶B+树： 与B树相比，

- 1、所有的非叶子节点只储存关键字信息；
- 2、所有的具体数据都存在叶子节点中；

3、所有的叶子节点之间都有一个链指针；

查询单个关键字 B树可以不到根结点就查询到；所以速度更快。但查询大于某个关键字或者小于某个关键字，B+树只需要沿着叶子节点链表遍历，B树还需要遍历根结点，所以B+树更快；

由于B+树非叶子节点不存储具体数据 所以每个节点B+树存储的数据更多，因此同样数据量的查询，B+树磁盘io读写次数更少。

因此 索引一般都用B+树。

单例模式 动态代理 快排算法 手写

负载均衡重试法和随机是啥

hash值存储nulltcpip协议 进程网络通信过程

如何保证代码的规范编写

(1) 类名首字母大写，字段、方法以及对象的首字母应该小写；

(2) 为了常规用途而创建一个类时，请采取"经典形式"，并包含对下述元素的定义：

```
equals()  
hashCode()  
toString()  
clone() (implement Cloneable)  
implement Serializable
```

(3) 使类尽可能短小精悍，而且只解决一个特定的问题。

(4) 让一切东西都尽可能地"私有"--private

(5) 避免使用"魔术数字"我们应创建一个常数，并为其使用具有说服力的描述性名称，并在整个程序中都采用常数标识符

(6)多加注释

常量的命名（全部大写，常加下划线）

常量的名字应该都使用大写字母，并且指出该常量完整含义。

类的命名（单词首字母大写）

包的命名（全部小写，由域名定义）

内核 也是程序 c语言 属于操作系统的一部分 直接调动我的硬件 操作

用户态 和我们自己的代码交互 内核去读该代码流