Java基础: 使用 final 修饰的 HashMap 是否可以 put 操作

final 修饰变量的值

HashMap 是引用类型, 值是对象的地址

所以只是地址不能变, 地址指向的对象可以进行 put 操作

Spring: IoC 和 DI

1. IoC: inversion of control, 控制反转

就是把原来代码里需要实现的对象创建和依赖，反转给容器来帮忙实现

我们需要创建一个容器, 同时需要一种描述来让容器知道要创建的对象与对象关系

这个描述最具体的表现就是我们所看到的配置文件

2. DI: Dependency Injection, 依赖注入

指对象被动接受依赖类而不自己主动去找, 不是从容器中查找它依赖的类, 而是在容器实例化对象时主动将它依赖的类注入给它

3. IoC 并不是 Sping 的特有功能, Spring 是支持 IoC 的容器之一, Spring 实现 IoC 的方式是 DI

数据库: union 和 union all

union: 自动去重, 默认按照第一个查询的第一列升序排序

union all: 包含全部结果, 不去重, 不排序

其他: 集群与分布式

集群是每台服务器都具有相同的功能, 处理请求时调用哪台服务器都可以, 主要作用是分流

分布式是将不同的业务放在不同的服务器中, 把一个请求拆分并交给多台服务器处理, 主要作用是提速

Java基础: 包装类和基本类型的区别

1. 包装类是对象, 可以为 null, 默认值是 null

2. 基本类型不能为 null, 有默认值(如: int 默认为 0)

3. 所以:

使用包装类作为参数, 可以避免参数为 null 时报错, 因为基本类型无法接收 null 值

实体类属性使用包装类可以避免没有值时被自动添加默认值

4. JDK1.5 以后 Java 增加了自动拆箱封箱功能, 包装类与基本类型可以使用 "==" 进行比较, 结果与两个基本类型比较一样

Spring: @Autowired 和 @Resource

1.

@Resource 是 Java 自己的注解, 不需要额外的 jar 支持, Spring 支持 @Resource

@Autowired 是 Spring 的注解, 在 org.springframework.beans 中

2.

默认情况下

@Resource 按照名称进行 bean 匹配, 如果找不到, 则按照类型匹配

@Autowired 按照类型进行 bean 匹配, 且不可以为 null(设置 required 为 false 可以允许为 null)

3.

@Resource 可以通过属性指定按照名称匹配(name=XXX)还是按照类型匹配(type=XXX), 如果两种属性都存在, 则必须都符合

@Autowired 配合 @Qualifier("XXX") 可以做到按照名称匹配, 并且可以解决按照类型匹配出多个依赖的情况

HTML/CSS: 两种隐藏页面元素的方法

1. display: none 隐藏, 并且不再占据页面空间

2. visibility：hidden 隐藏, 但是占据页面空间

其他: 缓存穿透和缓存雪崩

1. 缓存穿透

现象: 获取一个值为空的缓存, 因为没查到值而访问数据库, 导致每一次查询都会访问数据库

解决: 缓存时发现值本身就是空时, 将空设置为一个特定的值, 例如 -1

2. 缓存雪崩

现象: 同一批缓存的数据同一时间过期, 导致大量请求访问数据库

解决: 即便是同一个操作缓存的批量数据, 过期时间也要随机分配

Java基础: String 的比较为什么要使用 equals 而不能使用 ==

== 比较变量的值, 字符串作为引用类型, 变量值是对象所在的地址, 虽然字符串的内容一样, 但是所在地址可能不同

扩展1: 什么时候两个字符串内容相同但是地址不同

String a = new String("123"); 时, a 的地址并不是 "123" 的地址, 而是存放 "123" 的 String 对象的地址

扩展2: 为什么 "123" == "123" 是 ture

因为 "123" 本身是一个对象(可以"."出来方法), 前后两个 "123" 是同一个对象, 所以使用 == 时地址值相同得到 true

扩展3: 为什么 String a = new String("123"); a == "123" 是 false

因为扩展1和扩展2

扩展4: 为什么使用 equals 可以正确比较字符串

String 类重写了继承自 Object 的 equals 方法, 当参数不是 String 类型时直接返回 false, 当参数是 String 类型时, 对内容进行了判断

扩展5: 原始的 equals 方法

Object 的 equals 方法只有一行代码: return (this == obj);

数据库: 对于类型为 NUMBER 的字段, WHERE 的判断中使用 '0' 和 0 有什么区别

1. 对于查询结果没有影响

2. 使用 '0' 时, 数据库会进行隐式数据类型转换, 将 '0' 转为 0, 如果此列有索引, 会导致索引失效, 降低速度

3. 在索引列上做的任何操作, 如计算、函数、自动或手动的类型转换, 都会导致索引失效

数据库: 0.1 存入数据库后变成 .1

如果数据的列是字符类型, 而存储的 0.1 是数字时, 或者数字 0.1 与字符串拼接时, 0.1 就会变成 .1

因为数据库通过 to\_char 自动隐式转换数字时, 会忽略小数点前面的 0(尾部的 0 也会忽略), 变成 .1

扩展:

to\_char 本身是有两个参数的, 一个是要转换的内容, 另一个是转换的格式, 例如: to\_char(数字, '格式')

当内容为数字, 且没有格式参数时(数据库的隐式转换就没有格式参数), 默认就是忽略小数点前和末尾的 0

消息队列: 什么是消息队列, 消息队列有什么用

消息队列(Message queue), 是在消息的传输过程中保存消息的容器

队列是一种先进先出的数据结构, 消息队列就是把要传输的数据放在队列中依次执行

主要作用:

1. 解耦: 当 B C D 都要 A 的执行结果时, 传统做法是在 A 中分别调用 B C D, 一旦 B C D 发生变化, 或者有新的 E 也需要这个结果, A 就需要频繁改动

使用消息队列作为中间件接收 A 的结果, 任何需要使用这个结果的都自己从消息队列中获取, 这样产生结果的 A 就与使用这个结果的 B C D E 解耦了

2. 异步: 一个复杂的业务处理会有很多步骤, 甚至调用多个系统的接口, 复杂度的上升不可避免的导致响应速度变慢

将主要业务优先处理完成后, 后续根据主业务结果进行操作的连带业务由 MQ 来通知, 这样就能在主要业务执行完成后给与用户快速的响应

4. 削峰限流

控制从消息队列中获取消息的速度, 就把服务器从被动的不断处理请求中就解救出来了

变成了根据自身能力有计划的处理请求, 有效缓解服务器的压力

Spring: 面向切面(AOP)

1. AOP 是一种概念, Spring 只是实现了这种概念, 并不是发明和创造了 AOP

在传统的 OOP(面向对象) 开发中, 代码的逻辑是自上而下的线性执行, 在执行过程中, 会产生横切性问题

而这些横切性问题往往与业务逻辑关系不大, AOP 就是用来剥离代码中与业务无关的横切性问题

2. 什么是横切性问题:

当需要为多个不具有继承关系的对象引入同一个公共行为时(例如日志、安全检测等)

在 OOP 的线性执行中只能在每个对象中引用这个公共行为, 产生大量的重复代码, 不便于统一维护

3. AOP 是如何解决横切性问题的:

与 OOP 中调用一个具备所需公共行为的对象来做这件事不同, AOP 是让程序的某一类行为去触发这个公共行为

就像 JS 中的响应事件一样(例如: onclick、onchange等), AOP 可以在某一类行为的开始前、结束后或发生异常时, 自动去执行指定的公共行为

4. 举例说明 Spring 中的 AOP 应用:

Spring 的事务控制是基于 AOP 的

在 Spring 中, 想让一个方法内的数据库操作保持一个事务, 只需要在方法上标注 @Transactional 注解

Spring 依据 @Transactional 注解上的属性设置自动处理事务:

方法开始前自动开始事务, 执行结束后自动提交事务, 发生异常时自动回滚事务

5. Spring 是怎么做到 AOP 的: Spring 使用代理来实现 AOP

扩展一: 什么是代理, 有哪几种代理

代理就是创建一个代理对象来替换目标对象

理论上, 代理对象在拥有目标对象的一切功能前提下, 还有很多目标对象不具备的扩展功能

最主流的两种代理方式(不知道有没有其他的, 即便有也可以忽略):

1. JDK 动态代理

通过创建一个接口的实现类来完成对同样实现了此接口的目标对象的代理, 所代理的目标对象必须是某个接口的实现类

代理的大致过程:

编写的目标对象 B 实现了接口 A 的方法, 程序运行时以 B 作为模板创建代理对象 C, 同样实现了接口 A 的方法, 并在实现过程中添加指定的行为

在调用接口 A 的方法时, 其实调用的是 C 中已经添加了指定行为的方法, 而不是最开始编写的 B 的方法

2. CGLIB

与 JDK 动态代理类似, 不同点在于 CGLIB 创建的代理对象是目标对象的子类, 而不是同一个接口的另一个实现类

扩展二: Spring 与 AspectJ

Spring 和 AspectJ 都实现了 AOP , 但并不是一种语言

由于 AspectJ 的语法实在是太牛逼了, 所以 Spring 借鉴了 AspectJ 的语法, 但是底层原理是不一样的

Spring 用了 AspectJ 的哪些语法: 定位目标对象的语法, 也就是如何决定给哪些对象添加什么功能, 相当于 jQuery 选择器

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Java基础 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--------------------------------------------------

使用 final 修饰的 HashMap 是否可以 put 操作

final 修饰变量的值

HashMap 是引用类型, 值是对象的地址

所以只是地址不能变, 地址指向的对象可以进行 put 操作

--------------------------------------------------

包装类和基本类型的区别

1. 包装类是对象, 可以为 null, 默认值是 null

2. 基本类型不能为 null, 有默认值(如: int 默认为 0)

3. 所以:

使用包装类作为参数, 可以避免参数为 null 时报错, 因为基本类型无法接收 null 值

实体类属性使用包装类可以避免没有值时被自动添加默认值

4. JDK1.5 以后 Java 增加了自动拆箱封箱功能, 包装类与基本类型可以使用 "==" 进行比较, 结果与两个基本类型比较一样

--------------------------------------------------

String 的比较为什么要使用 equals 而不能使用 ==

== 比较变量的值, 字符串作为引用类型, 变量值是对象所在的地址, 虽然字符串的内容一样, 但是所在地址可能不同

扩展1: 什么时候两个字符串内容相同但是地址不同

String a = new String("123"); 时, a 的地址并不是 "123" 的地址, 而是存放 "123" 的 String 对象的地址

扩展2: 为什么 "123" == "123" 是 ture

因为 "123" 本身是一个对象(可以"."出来方法), 前后两个 "123" 是同一个对象, 所以使用 == 时地址值相同得到 true

扩展3: 为什么 String a = new String("123"); a == "123" 是 false

因为扩展1和扩展2

扩展4: 为什么使用 equals 可以正确比较字符串

String 类重写了继承自 Object 的 equals 方法, 当参数不是 String 类型时直接返回 false, 当参数是 String 类型时, 对内容进行了判断

扩展5: 原始的 equals 方法

Object 的 equals 方法只有一行代码: return (this == obj);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Spring \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--------------------------------------------------

IoC 和 DI

1. IoC: inversion of control, 控制反转

就是把原来代码里需要实现的对象创建和依赖，反转给容器来帮忙实现

我们需要创建一个容器, 同时需要一种描述来让容器知道要创建的对象与对象关系

这个描述最具体的表现就是我们所看到的配置文件

2. DI: Dependency Injection, 依赖注入

指对象被动接受依赖类而不自己主动去找, 不是从容器中查找它依赖的类, 而是在容器实例化对象时主动将它依赖的类注入给它

3. IoC 并不是 Sping 的特有功能, Spring 是支持 IoC 的容器之一, Spring 实现 IoC 的方式是 DI

--------------------------------------------------

面向切面(AOP)

1. AOP 是一种概念, Spring 只是实现了这种概念, 并不是发明和创造了 AOP

在传统的 OOP(面向对象) 开发中, 代码的逻辑是自上而下的线性执行, 在执行过程中, 会产生横切性问题

而这些横切性问题往往与业务逻辑关系不大, AOP 就是用来剥离代码中与业务无关的横切性问题

2. 什么是横切性问题:

当需要为多个不具有继承关系的对象引入同一个公共行为时(例如日志、安全检测等)

在 OOP 的线性执行中只能在每个对象中引用这个公共行为, 产生大量的重复代码, 不便于统一维护

3. AOP 是如何解决横切性问题的:

与 OOP 中调用一个具备所需公共行为的对象来做这件事不同, AOP 是让程序的某一类行为去触发这个公共行为

就像 JS 中的响应事件一样(例如: onclick、onchange等), AOP 可以在某一类行为的开始前、结束后或发生异常时, 自动去执行指定的公共行为

4. 举例说明 Spring 中的 AOP 应用:

Spring 的事务控制是基于 AOP 的

在 Spring 中, 想让一个方法内的数据库操作保持一个事务, 只需要在方法上标注 @Transactional 注解

Spring 依据 @Transactional 注解上的属性设置自动处理事务:

方法开始前自动开始事务, 执行结束后自动提交事务, 发生异常时自动回滚事务

5. Spring 是怎么做到 AOP 的: Spring 使用代理来实现 AOP

扩展一: 什么是代理, 有哪几种代理

代理就是创建一个代理对象来替换目标对象

理论上, 代理对象在拥有目标对象的一切功能前提下, 还有很多目标对象不具备的扩展功能

最主流的两种代理方式(不知道有没有其他的, 即便有也可以忽略):

1. JDK 动态代理

通过创建一个接口的实现类来完成对同样实现了此接口的目标对象的代理, 所代理的目标对象必须是某个接口的实现类

代理的大致过程:

编写的目标对象 B 实现了接口 A 的方法, 程序运行时以 B 作为模板创建代理对象 C, 同样实现了接口 A 的方法, 并在实现过程中添加指定的行为

在调用接口 A 的方法时, 其实调用的是 C 中已经添加了指定行为的方法, 而不是最开始编写的 B 的方法

2. CGLIB

与 JDK 动态代理类似, 不同点在于 CGLIB 创建的代理对象是目标对象的子类, 而不是同一个接口的另一个实现类

扩展二: Spring 与 AspectJ

Spring 和 AspectJ 都实现了 AOP , 但并不是一种语言

由于 AspectJ 的语法实在是太牛逼了, 所以 Spring 借鉴了 AspectJ 的语法, 但是底层原理是不一样的

Spring 用了 AspectJ 的哪些语法: 定位目标对象的语法, 也就是如何决定给哪些对象添加什么功能, 相当于 jQuery 选择器

--------------------------------------------------

@Autowired 和 @Resource

1.

@Resource 是 Java 自己的注解, 不需要额外的 jar 支持, Spring 支持 @Resource

@Autowired 是 Spring 的注解, 在 org.springframework.beans 中

2.

默认情况下

@Resource 按照名称进行 bean 匹配, 如果找不到, 则按照类型匹配

@Autowired 按照类型进行 bean 匹配, 且不可以为 null(设置 required 为 false 可以允许为 null)

3.

@Resource 可以通过属性指定按照名称匹配(name=XXX)还是按照类型匹配(type=XXX), 如果两种属

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 数据库 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--------------------------------------------------

union 和 union all

union: 自动去重, 默认按照第一个查询的第一列升序排序

union all: 包含全部结果, 不去重, 不排序

--------------------------------------------------

对于类型为 NUMBER 的字段, WHERE 的判断中使用 '0' 和 0 有什么区别

1. 对于查询结果没有影响

2. 使用 '0' 时, 数据库会进行隐式数据类型转换, 将 '0' 转为 0, 如果此列有索引, 会导致索引失效, 降低速度

3. 在索引列上做的任何操作, 如计算、函数、自动或手动的类型转换, 都会导致索引失效

--------------------------------------------------

0.1 存入数据库后变成 .1

如果数据的列是字符类型, 而存储的 0.1 是数字时, 或者数字 0.1 与字符串拼接时, 0.1 就会变成 .1

因为数据库通过 to\_char 自动隐式转换数字时, 会忽略小数点前面的 0(尾部的 0 也会忽略), 变成 .1

扩展:

to\_char 本身是有两个参数的, 一个是要转换的内容, 另一个是转换的格式, 例如: to\_char(数字, '格式')

当内容为数字, 且没有格式参数时(数据库的隐式转换就没有格式参数), 默认就是忽略小数点前和末尾的 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 消息队列 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

什么是消息队列, 消息队列有什么用

消息队列(Message queue), 是在消息的传输过程中保存消息的容器

队列是一种先进先出的数据结构, 消息队列就是把要传输的数据放在队列中依次执行

主要作用:

1. 解耦: 当 B C D 都要 A 的执行结果时, 传统做法是在 A 中分别调用 B C D, 一旦 B C D 发生变化, 或者有新的 E 也需要这个结果, A 就需要频繁改动

使用消息队列作为中间件接收 A 的结果, 任何需要使用这个结果的都自己从消息队列中获取, 这样产生结果的 A 就与使用这个结果的 B C D E 解耦了

2. 异步: 一个复杂的业务处理会有很多步骤, 甚至调用多个系统的接口, 复杂度的上升不可避免的导致响应速度变慢

将主要业务优先处理完成后, 后续根据主业务结果进行操作的连带业务由 MQ 来通知, 这样就能在主要业务执行完成后给与用户快速的响应

4. 削峰限流

控制从消息队列中获取消息的速度, 就把服务器从被动的不断处理请求中就解救出来了

变成了根据自身能力有计划的处理请求, 有效缓解服务器的压力

--------------------------------------------------

Spring: 事务失效

现象:

在同一个 service 中, 有两个方法 A 和 B

A 和 B 都使用 @Transactional 标注为需要开启新的独立事务

A 方法插入一条数据, 然后调用 B 方法插入一条数据

如果 A 方法在调用了 B 以后发生异常, A 和 B 插入的数据都将回滚

也就是 B 方法设置的新开一个事务的注解失效了

原理:

Spring 的事务控制基于 AOP, AOP 基于代理

调用 service 的 A 方法时, 之所以能自动开始事务, 是因为真实调用的是 service 的代理对象的 A 方法

代理对象 A 方法开启新的事务之后调用了原对象的 A 方法, 所以完成了事务控制

当在 A 方法中直接调用 B 方法时, 其实调用的 this.B 方法

这个 this 指向的是没有代理的原 service, 所以没有开启事务

解决办法:

设置 @EnableAspectJAutoProxy 的 exposeProxy 属性设置为 true(默认为 false)

再通过静态方法 AopContext.currentProxy() 方法强制在原 service 的 A 方法中获取到代理对象的 B 方法, 可以解决这个问题

--------------------------------------------------

Redis: Redis 是什么

Redis 的全称是 REmote Dictionary Server, 是一个基于键值对(key-value)的 noSQL 数据库, 开源. 由于 Redis 的数据存储在内存中, 所以读写效率极高

Redis 也支持数据的持久化, 从而避免因为断电或者机械故障导致内存中的数据丢失的问题

Redis 官方并不支持 Windows 操作系统, Windows 版本的 Redis 并不一定能完全兼容 Redis 的所有功能. 真正的企业应用应该使用 Linux 系统

Redis 的 8 个特性: 速度快; 基于键值对的数据结构服务器; 功能丰富; 简单稳定; 客户端语言多; 持久化; 主从复制; 支持高可用和分布式

Redis 是单线程的

扩展: 单线程的优缺点

1. 单线程可以简化数据结构和算法的实现

2. 单线程避免了线程切换和竞态产生的消耗, 对于服务端开发来说, 锁和线程切换通常是性能杀手

3. 单线程的问题: 对于每个命令的执行时间是有要求的. 如果某个命令执行过长, 会造成其他命令的阻塞

这对于 Redis 这种高性能的服务来说是致命的, 所以 Redis 是面向快速执行场景的数据库

--------------------------------------------------

Redis: 数据结构之字符串

字符串类型是 Redis 最基础的数据结构. 所有键都是字符串类型, 而且其他几种数据结构都是在字符串类型基础上构建的.

字符串类型的值实际可以是字符串(简单的字符串、复杂的字符串(JSON、XML))、数字(整数、浮点数), 甚至是二进制(图片、音频、视频), 但是值最大不能超过 512MB

字符串类型的内部编码有3种, Redis 会根据不同情况来决定使用哪种内部编码实现:

1. int: 8个字节的长整型

2. embStr: 小于等于 39 个字节的字符串

3. raw: 大于 39 个字节的字符串

字符串类型的经典使用场景:

1. 缓存: 数据放在 Redis 中读取速度快, 持久化改为定期执行来分散数据库的压力. 对于一些常用的数据, 可以极大地提高效率和节省数据库资源

2. 计数器: 使用 incr 每次自增并返回自增后的值

3. 限速: 通过设置 expire, 配合 setNx 可以限制指定时间内的执行次数

4. 分布式系统中共享 session: 每个服务器都从 Redis 中获取 session

--------------------------------------------------

Spring: @Scope

用于描述 Spring 容器新建 Bean 的方式

常用属性值:

1. @Scope("singleton") 一个 Spring 容器中只有一个 Bean 的实例, 当没有使用 @Scope 时, 这是默认配置

2. @Scope("prototype") 每次调用都新建一个 Bean 实例

3. @Scope("request") 每个 http request 新建一个 Bean 实例

4. @Scope("session") 每个 http session 新建一个 Bean 实例

扩展: @Scope("prototype") 失效

现象:

有 A 和 B 两个 Bean, A 依赖 B

当 A 设置为 @Scope("singleton"), 而 B 设置为 @Scope("prototype") 时

本意是 A 只初始化一次, B 每次使用时都初始化一个新的

但是由于 A 只初始化一次, 所以 A 中的 B, 也只会注入一次, 导致每次 getB() 获取的都是同一个 B

解决办法:

1. 在 getB() 方法中通过 ApplicationContext 每次都获取一个新的 B 返回

2. 更优雅的解决办法: @Lookup

将 getB() 方法设置为 abstract 方法, 并在方法上添加 @Lookup 注解就可以让 getB() 每次都返回一个新的 B

@Lookup 的实现原理是代理

--------------------------------------------------

前端基础: 引入 CSS 样式的两种方式和区别

1. <link rel="stylesheet" type="text/css" href="地址"/>

2. <style type="text/css">@import '地址'</style>

区别:

1. link 属于 HTML 标签, 无兼容问题, 而 @import 是 CSS 提供的, 只在 IE5 以上才能识别

2. 页面被加载的时 link 会同时被加载, @import 引用的 CSS 会等到页面被加载完再加载

--------------------------------------------------

Redis: 数据结构之哈希

哈希是指键的值本身是一个键值对, 映射关系叫 field-value

哈希的内部编码有两种:

1. zipList(压缩列表):

当哈希类型元素个数小于 hash-max-zipList-entries 配置(默认 512 个), 同时所有值都小于 hash-max-zipList-value 配置(默认 64 字节)时

Redis 会使用 zipList 作为哈希的内部实现

zipList 使用更加紧凑的结构实现多个元素的连续存储, 所以在节省内存方面比 hashtable 更加优秀

使用哈希时, 要尽可能的让 Redis 使用 zipList 的内部实现, 节省大量资源

2. hashtable(哈希表):

当哈希类型无法满足 zipList 的条件时, Redis 会使用 hashtable 作为哈希的内部实现, 因为此时 zipList 的读写效率会下降

哈希可以模仿关系型数据的表结构, 但是需要注意的是哈希类型和关系型数据库有两点不同之处:

1. 哈希类型是稀疏的, 而关系型数据库是完全结构化的.

哈希类型每个键可以有不同的 field, 而关系型数据库一旦添加新的列, 所有行都要为其设置值(即使为 null)

2. 关系型数据库可以做复杂的关系查询, 而 Redis 去模拟关系型复杂查询开发困难, 维护成本高

哈希类型的经典实用场景: 存储关系数据库的整条数据

--------------------------------------------------

数据库: LEFT JOIN 时, 条件放在 ON 和 WHERE 的区别

例子:

SELECT \* FROM 左表 LEFT JOIN 右表 ON 关联关系 AND P1 WHERE P2

1. P1 的作用:

P1 是对右表的筛选, 无法影响左表的任何一条数据, 无论 P1 筛选的是筛选左表还是右表, 最终结果集都会包含左表所有的结果

2. 如果 P2 筛选的是右表

会筛选掉在右表没有匹配的左表数据, 因为此时右表无匹配的数据所有右表值都为 NULL(除非是 IS NULL)

总结: 右表的筛选条件要放在 ON 后面, 左表的筛选条件要放在 WHERE 后面

--------------------------------------------------

前端基础: jQuery 中 $().attr() 与 $().prop() 的区别

1. attr:

操作的是 HTML 对象, 也就是标签内容中各个属性键值对中""中间的内容

这是一种文本直接取值, ""中间是什么, 获取到的就是什么

包括但不仅限于 HTML 标准中规定的属性, 只要是写在标签上或添加到标签上的都可以操作

attr 本身也可以像 HTML 添加自定义的属性键值对

2. prop:

操作的是 DOM 对象, DOM 是 HTML 标签的抽象, 浏览器根据 HTML 标签构建每个标签的 DOM 对象

无论标签中是否显式的写出, DOM 对象都拥有这个标签在 HTML 标准中的所有属性和方法

在将 HTML 标签转为 DOM 对象的过程中, 属性值并不完全是等值替换的

例如: <input id="abc" type="radio" checked>

$("#abc").attr("checked"); 获取到的是 HTML 标签上的文本值 "checked"

$("#abc").prop("checked"); 获取到的是 DOM 对象中的属性值 true

注意1: DOM 对象只是根据 HTML 标签内容获取属性值, 并不是"HTML 标签是 DOM 的标准", 而是"DOM 是 HTML 标签 的标准"

注意2: 使用 attr 对 HTML 标签一些特殊属性值的操作是完全无效的

例子中的 checked 属性就是, 只有使用 prop 修改 DOM 属性值的 true 和 false 才能切换选中和不选中, 操作 HTML 上的文本是无效的

--------------------------------------------------

Redis: 数据结构之列表

用来存储多个有序的字符串, 列表中的每个字符串称为元素(element), 一个列表最多可以存储 4294967295(2 的 32 次方 - 1) 个元素

列表可以两端插入(push)和弹出(pop), 还可以获取指定范围的元素列表、获取指定索引下标的元素等.

列表是一种比较灵活的数据结构, 它可以充当栈和队列的角色. 在实际开发上有很多应用场景

列表类型的两个特点:

1. 列表中的元素是有序的, 这就意味着可以通过索引下标获取某个元素或者某个范围内的元素列表

2. 列表中的元素可以是重复的

列表的内部编码: zipList 和 linkedList, 无法满足 zipList 时使用 linkedList

Redis3.2 版本提供了 quickList 内部编码, 简单地说它是以一个 zipList 为节点的 linkedList

它结合了 zipList 和 linkedList 两者的优势, 为列表类型提供了一种更为优秀的内部编码实现

它的设计原理可以参考 Redis 的一个作者的博客：https://matt.sh/redis-quicklist

列表类型的经典实用场景:

1.消息队列&负载均衡: 列表类型的数据通过 lPush + brPop 组合即可实现阻塞队列

生产者客户端使用 lrPush 从列表左侧插入元素, 多个消费者客户端使用 brPop 命令阻塞式的“抢”列表尾部的元素, 多个客户端保证了消费的负载均衡和高可用性

利用取范围功能, 列表类型还可以作为翻页列表使用

2. lPush + lPop = Stack (栈)

3. lPush + rPop = Queue(队列)

4. lPush + lTrim = Capped Collection (有限集合)

5. lPush + brPop = Message Queue (消息队列)

--------------------------------------------------

Java基础: BigDecimal 也会丢失精度

使用 BigDecimal 时, 只有 new BigDecimal(String) 能保持精度

如果传入的参数是其他类型, 依然会丢失精度

例1: 使用浮点型参数构造 BigDecimal 对象, 无法保证精度

BigDecimal a = new BigDecimal(1.01);

BigDecimal b = new BigDecimal(1.02);

System.out.println(a.add(b));

输出：

2.0300000000000000266453525910037569701671600341796875

例2: 同样的数值, 换成 String 型参数构造 BigDecimal 对象可以保证精度

BigDecimal c = new BigDecimal("1.01");

BigDecimal d = new BigDecimal("1.02");

System.out.println(c.add(d));

输出：

2.03

--------------------------------------------------

数据库: 事务

什么是事务

事务是应用程序中一系列严密的操作且所有操作必须成功完成, 任何一个操作的失败都会让所有更改撤销

事务的 4 个特性: ACID

1. Atomicity 原子性: 要么全部成功, 要么全部失败

2. Consistency 一致性: 数据库只包含成功事务提交的结果, 无论事务因何中断, 都不能将一部分内容写入数据库

3. Isolation 隔离性: 事务内部的操作及使用的数据对其他并发事务是隔离的, 并发执行的各个事务之间不能互相干扰

4. Durability 持续性(永久性): 事务提交后对数据的改变应该是永久的, 接下来的其它任何操作或故障不该对其执行结果有影响

隔离级别

1. Read Uncommitted 读取未提交内容: 当前事务可以看到其他事务未提交的执行结果

2. Read Committed 读取提交内容(Oracle 的默认隔离级别): 事务进行过程中可以读取其他事务已经提交的执行结果

3. Repeatable Read 可重读(MySQL 的默认隔离级别): 锁住正在操作的数据, 保证事务内操作的数据前后一致

4. Serializable 可串行化: 最高隔离级别, 通过强制事务排序让事务之间不发生冲突, 可能会导致大量的超时现象和锁竞争

扩展: 因隔离级别导致的问题

1. 脏读(Dirty Read), 隔离级别为 Read Uncommitted 时会出现

事务 A 更新了数据库并且未提交, 事务 B 在读取并使用了 A 未提交的数据, A 失败回滚导致 B 使用了错误的数据

2. 不可重复读(Non-Repeatable Read), 隔离级别为 Read Uncommitted 或 Read Committed 时会出现

事务 A 获取到了一个值, 此时事务 B 更改了这个值并提交, 当 A 再次获取这个值时发现与之前的数据不一致了

3. 幻读(Phantom Read) 隔离级别为 Read Uncommitted 或 Read Committed 或 Repeatable Read 时会出现

事务 A 通过 WHERE 条件更新一批数据并且未提交, 事务 B 插入一条符合 WHERE 条件的新数据并提交, 此时 A 提交事务以后, 发现这条新数据并没有更新

注意: 虽然 Repeatable Read 时会锁住要操作的数据, 但 B 是新插入一条, 这条数据并不在锁住的范围内, 所以也会出现幻读

--------------------------------------------------

Redis: 数据结构之集合

用来保存多个字符串元素.

与列表类型不一样的是集合中不允许有重复元素, 并且集合中的元素是无序的, 不能通过索引下标获取元素.

一个集合最多可以存储 4294967295(2 的 32 次方 - 1) 个元素

Redis 除了支持集合内的增删改查, 同时还支持多个集合取交集、并集、差集, 合理地使用好集合类型能在实际开发中解决很多实际问题

集合的内部编码类型有两种:

集合中所有元素都是整数, 且数量小于 set-max-intSet-entries 配置(默认 512 个)时, 使用 intSet, 不满足条件时使用 hashTable

集合类型的经典实用场景:

1. 标签:

一个用户可能对娱乐、体育比较感兴趣, 另一个用户可能对历史、新闻比较感兴趣, 这些兴趣点就是标签

有了这些数据就可以得到喜欢同一个标签的人以及用户的共同标签, 这些数据对于用户体验以及增强用户黏度比较重要

例如一个电子商务的网站会对不同标签的用户做不同类型的推荐, 比如对数码产品比较感兴趣的人

在各个页面或者通过邮件的形式给他们推荐最新的数码产品, 通常会为网站带来更多的利益

2. sAdd = Tagging(标签)

3. sPop/sRandMember = Random item(生成随机数, 比如抽奖)

4. sAdd + sInter = Social Graph(社交需求)

--------------------------------------------------

Spring: @ControllerAdvice 控制器建言

类注解, 通过 @ControllerAdvice, 可以在一个地方对所有控制器进行管理

通常配合以下三种方法注解使用

1. @ExceptionHandler: 处理控制器里的异常

在方法添加了此注解后, 会在控制器抛出异常时自动被调用

通过设置 @ExceptionHandler 的 value 属性, 可以自定义处理的异常类型

被标记的方法只需要在方法定义中声明所需的参数, Spring 就会在调用时自动注入

扩展: 除了控制器发生的是什么异常以外, 最常用的参数是 HandlerMethod

HandlerMethod 对象内包含发生异常的控制器和方法的所有信息

实例:

/\*\*

\* 处理所有控制器抛出的异常

\*

\* @param exception 控制器抛出的未处理的异常

\* @param handlerMethod 抛出异常的控制器及方法的信息

\*/

@ExceptionHandler(value = Exception.class) // 通过 value 设置此方法处理的异常类型

public void exception(Exception exception, HandlerMethod handlerMethod) {

System.out.println(handlerMethod.getMethod().getName() + " 发生了异常: " + exception.getMessage());

// 返回值不仅可以是 void 的, 还可以是 ModelAndView, 或其他任意类型

// 如果想在处理异常后继续返回数据, 还要像普通的控制器方法一样添加 @ResponseBody

}

2. @InitBinder 用于数据的绑定规则

前端传入控制器的数据, 会根据控制器的参数类型和名称自动绑定, 绑定过程由 Spring 自动处理

@InitBinder 注解标注的方法可以让开发人员参与到绑定的过程中, 进行一些自定义的处理

@InitBinder 标注的方法会自动获得 Spring 注入的 WebDataBinder 对象, 用于注册各种属性编辑器

例: 使用 Spring 自带的属性编辑器 CustomDateEditor (除了 CustomDateEditor 以外还有很多, 不一一列举了)

如果前端传入日期格式的字符串, 而控制器的方法中使用 Date 接收, Spring 默认是不支持的, 所以要自己定义

/\*\*

\* 自定义 String 转为 Date 的规则

\*

\* @param binder 编辑器的注册器

\*/

@InitBinder

public void stringToDate(WebDataBinder binder) {

// 第一个参数定义转换方法

// 第二个参数定义是否允许 null 值

CustomDateEditor editor = new CustomDateEditor(

new SimpleDateFormat("yyyy-MM-dd hh:mm:ss"), true

);

// 注册编辑器, 并设置此编辑器用于控制器中哪种类型的参数

binder.registerCustomEditor(Date.class, editor);

}

扩展: 自定义编辑器

让一个类继承 org.springframework.beans.propertyeditors.PropertiesEditor 就可以成为一个编辑器

3. @ModelAttribute 太复杂, 而且大多数功能可能到退休都用不上, 这里就只举一个简单的例子

/\*\*

\* 给所有请求添加一个参数

\*

\* @param model 这个对象可以单独写一篇了, 暂时就不详解了

\*/

@ModelAttribute

public void addParam(Model model) {

// 此处添加 test 属性, 可以在作用范围内的控制器的所有方法中通过参数声明获取到

model.addAttribute("test", "abc");

}

特别注意:

@ExceptionHandler @InitBinder @ModelAttribute

在 @Controller 中的方法上使用, 作用范围是当前控制器

在 @ControllerAdvice 中的方法上使用, 作用范围是所有控制器

--------------------------------------------------

Redis: 数据结构之有序集合

有序集合保留了集合不能有重复成员的特性, 不同的是元素可以排序. 和列表使用索引下标作为排序依据不同, 它给每个元素设置一个分数(score)作为排序的依据

有序集合提供了获取指定分数和元素范围查询、计算成员排名等功能

元素不能重复, 但是分数可以重复. 就像一个班级里学号不能重复, 但是成绩可以相同

相对于集合来说, 增加了排序功能的代价是性能上的降低

有序集合的内部编码有两种

1. zipList: 当有序集合的元素个数小于 zSet-max-zipList-entries 配置(默认 128 个), 同时每个元素的值都小于 zSet-max-zipList-value 配置(默认 64 字节)

2. skipList: 当 zipList 条件不满足时, 有序集合会使用 skipList 作为内部实现, 因为此时 zipList 的读写效率会下降

有序集合类型的经典实用场景: 排行榜系统