技术规范

> 以科技为主导的互联网企业, 都离不开技术的平台化和规范性。
>
> 本文将详细介绍货拉拉-卖车及车后大市场业务部技术行为规范。

导读：本文只涉及后端研发、前端、APP、运维、产品、测试的同学。
目的：引导团队成员统一技术行为规范，提高沟通效率，理解效率和开发效率以及流程效率。

日期	版本	编著
2020-08-20	1.0-Beta	Timi

〇、全局规范

0.1 取名规范

- 应当采用可拼读英文单词作为主体, 优先遵从lowerCamelCase (小驼峰命名法), 提高主体识别性。
- 不应该出现中英混写, 拼音和英文混写以及不规范缩写的情况。
- 不应该使用平台关键字、预留关键字和特殊关键字以及歧义字。

0.2 代码规范

- Java开发的代码编写应该优先遵从 阿里规约 规范要求, 对标国际编码水平。

一、开发规范

1.1 工程管理(All)

1.1.1 工程类型

- 构建工程可选Maven、Gradle等大众化主流构建工具, 推荐使用Maven。
- IDE开发工具可选 后端:IDEA、Eclipse; 前端:WebStorm、HBuilder等, 推荐使用IDEA和WebStorm。
- 项目较大时应该建设父子工程结构, 父工程不实现具体业务, 具体业务需要清晰的分布在子工程中(~~子工程根据业务划分~~)。

1.1.2 工程名称

- 工程命格式: 工程性质-应用名称
- 工程名书写: 工程性质和应用名称均为英文单词。
- 工程名组成: 英文字母 与 "-" (仅用于分隔工程性质和应用名称)。**不支持** 数字, 中文, 特殊字符 **等**..

```
  工程性质：hll
  应用名称：crm(客户关系管理系统)、oms(订单管理系统)、charge(特惠充电项目)、ins(惯性导航系统)、ams(活动管理系统)..
```

- 如果业务复杂，有父子工程请参考以下命名和结构。

```
hll-crm
├- crm-c
|   └- src
|   |   └- main
|   |   |   └- java
|   |   |   └- sources
|   |   |   └- webapp
|   |   └- test
|   |   |   └- java
|   |   |   └- sources
├- crm-r
├- crm-m
├- pom.xml
├- README.md
```

1.1.3 工程依赖

- 线上应用不要依赖 SNAPSHOT 版本（安全包除外）。
- 依赖于一个二方库群时，必须定义一个统一的版本变量，避免版本号不一致。

> 依赖 springframework-core,-context,-beans，它们都是同一个版本。
>
> 可以定义一 个变量来保存版本：${spring.version}，定义依赖的时候，引用该版本。

- 引入或升级依赖前必须通过评估和验证。
- 二方库的新增或升级，保持除功能点之外的其它 jar 包仲裁结果不变。
- 禁止在子项目的 pom 依赖中出现相同的 GroupId，相同的 ArtifactId，但是不同的 Version。
- 为避免应用二方库的依赖冲突问题，二方库发布者应当遵循以下原则：

> 1）精简可控原则。移除一切不必要的 API 和依赖，只包含 Service API、必要的领域模型对象、Utils 类、常量、枚举等。如果依赖其它二方库，尽量是 provided 引入，让二方库使用者去依赖具体版本号；无 log 具体实现，只依赖日志框架。
>
> 2）稳定可追溯原则。每个版本的变化应该被记录，二方库由谁维护，源码在哪里，都需要能方便查到。除非用户主动升级版本，否则公共二方库的行为不应该发生变化。

1.2 编码规范(Java)

1.2.1 命名规范

包命名

包命名: cn.huolala[.工程名.应用名](.子应用名)

包组成: 小写字母+数字, 遇多单词以`.`拆分。

正例	反例	提示
cn.huolala.crm.c	cn.huolala.crm.crm.c	应用名重复
cn.huolala.crm.c.order	cn.huolala.crm.cOrder	多单词应以 . 拆分
cn.huolala.crm.c.order.controller	cn.huolala.crm.c.order.Controller	报名只能包含小写字母和数字
cn.huolala.crm.c.order.controller2	cn.huolala.crm.c.order.2controller	首字符必须是小写字母

类命名

普通类

类命名: 遵从UpperCamelCase (大驼峰命名法), 首字母大写。
子类名: 子类业务名称+父类名。

```
- 父类: class Order
- 子类: class PresellOrder extends Order
```

实现类

实现类命名规则: 接口命+Impl

```
- 接口: interface OrderServer
- 抽象类: abstract class AbstractOrderServer implements OrderServer
- 实现类: class OrderServerImpl implements OrderServer
```

设计类

类中使用了设计模式, 应该在类名上追加设计模式名称。

```
public class OrderFactory;
public class LoginProxy;
public class ResourceObserver;
```

特殊类

- Exception: 异常类名必须以Exception结尾, 且应统一放置在exception包下
- Enum: 枚举类名必须以Enum结尾, 且应统一放置在enums包下
- Test: 测试类名必须已Test结尾, 且应统一放置在测试包下: src/test/java..
- Kit: 与业务无关的工具类名必须以Kit结尾, 且应统一放置在kit包下
- component: 与业务相关的封装类以Util结尾, 应统一放置在component包下
- 常量类: 常量类名必须以Constant结尾, 且应统一放置在constant包下
- **Do / Bo / Dto / Vo / Ao / Po等** 同样遵守驼峰命名法。

方法名

- 必须遵从lowerCamelCase (小驼峰命名法) 。
- 任何get/set方法必须以get/set+变量名的风格, 不可出现`isActive()`类似名称。

```
1） 获取单个对象的方法用get作前缀。
2） 获取多个对象的方法用list作前缀。
3） 获取统计值的方法用count作前缀。
4） 插入的方法用save/add/insert作前缀。
5） 删除的方法用del/delete作前缀。
6） 修改的方法用update作前缀。
7） 根据主键执行新增/修改的方法用modify作前缀。
```

- 类内方法定义的顺序依次是：公有方法或保护方法 > 私有方法 > getter/setter方法 > equals/hashCode方法 > toString方法。

变量名

必须遵从lowerCamelCase (小驼峰命名法) 。

常量名

必须全部大写, 多单词之间采用`_`分隔。

数据库

- 数据库名称格式必须是: db[_lala]_app
- > 其中app对应应用名称, `[]`中的内容根据是否外部业务来决定, 比如CRM系统所用的数据库则不加_lala: db_crm;
  >
  > 而支付系统是给消费者使用的则需要加lala: db_lala_pay。
- 表名、字段名必须使用小写字母或数字，禁止出现数字开头，禁止两个下划线中间只 出现数字。

  > 数据库字段名的修改代价很大，因为无法进行预发布，所以字段名称需要慎重考虑。
- 数据库表名格式必须是: t_app_business[_assistant1][_assistant2]..

  > 其中app对应应用名称, business则是对应的业务/功能/模块名称
  >
  > `[]`中的内容根据是否是某个业务的副表来决定, 比如CRM系统的"申请表"则命名为: t_crm_order;
  >
  > 而其副业务收款信息表则命名: t_crm_order_payments;
  >
  > 若收款信息仍然有副业务收款明细则命名为: t_crm_order_payments; 以此类推..
- 数据库表字段建设顺序要求: 主键 > 外键 > 业务字段 > 冗余字段 > 常规字段。

  > 其中固定字段如: 版本、状态/可见/删除、创建时间、创建人、修改时间、修改人等
- 表名不使用复数名词。
- 禁用保留字，如 desc、range、match、delayed 等，请参考 MySQL 官方保留字。
- 主键索引名为 pk*字段名；唯一索引名为 uk*字段名；普通索引名则为 idx_字段名。

  > 说明：pk_ 即 primary key；uk_ 即 unique key；idx_ 即 index 的简称。
- 小数类型为 decimal，禁止使用 float 和 double。

> float 和 double 在存储的时候，存在精度损失的问题，很可能在值的比较时，得到不 正确的结果。
>
> 如果存储的数据范围超过 decimal 的范围，建议将数据拆成整数和小数分开存储。

- varchar 是可变长字符串，不预先分配存储空间，长度不要超过 5000。

> 如果存储长 度大于此值，定义字段类型为 text，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

- 表必备三字段：id, create_at, create_by, update_at, update_by, delete_tag。

> 说明：其中 id 必为主键，类型为 bigint unsigned、单表时自增、步长为 1。
>
> create_at, update_at的类型均为 datetime 类型，前者现在时表示主动创建，后者过去分词表示被动更新。
>
> create_by，update_by, 的类型均为bigint类型, delete_tag 的类型均为bit类型。

- 如果修改字段含义或对字段表示的状态追加时，需要及时更新字段注释。
- 字段允许适当冗余，以提高查询性能，但必须考虑数据一致。冗余字段应遵循：

```
1）不是频繁修改的字段。
2）不是 varchar 超长字段，更不能是 text 字段。
```

- 单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。

字段表

> 建议在数据库中遇到如下含义的字段使用特定的字段名称:

字段名	含义
sn	当前申请单流水号（唯一标识码）
status	当前表状态必须命名为status
amount	金额
apply_date	申请日期
approval_id	审核流id
com_id	分公司id
handler	操作人
handler_by	处理人ID
mobile	手机号码(需要脱敏)
telephone	固定电话（例如：0755-38888888）
tenancy	租期
idcard	证件号码(需要脱敏)
fee	单笔钱（gpsFee【单笔GPS费】，licenseFee【上牌费】，）
member	会员
affiliated	挂靠
referrer	推荐人（老带新的推荐人）
tax	税，税收，税务
versions	版本号
create_at	新增时间
create_by	新增人
update_at	修改时间
update_by	修改人
delete_tag	逻辑删除
ts	防并发时间戳

缩写表

> 建议在需要简写的场景中遇到如下含义的词汇采用特定缩写:

缩写	全拼	含义
app	application	应用程序
avg	average	平均
bg	background	背景
conn	connection	连接（数据库连接）
ctrl	control	控制
ctx	context	上下文信息
dao	Data AccessObject	数据访问对象
del	delete	删除、移除
doc	document	文档、文件
err	error	错误
esc	escape	退出
ex	exception	异常
inc	increment	增量、增长
info	information	信息、消息
init	initial	初始化
img	image	图片
lab	label	标签
len	length	长度
lst	list	集合、列表
lib	library	库
mgr	manager	管理者，经理
msg	message	消息
pic	picture	图片
pos	position	职位
pwd	password	密码
src	source	来源、出处
str	string	字符串
sys	system	系统
win	window	窗口
veh	vehicle	车辆

缩写	全拼	含义
sc	salesclue	线索
llp	la la partner	汽销合作模式
stf	staff	员工
addr	Address	地址
com	company	分公司

1.2.2 格式规范

大括号

- 左大括号前不换行。
- 左大括号后换行。
- 右大括号前换行。
- 表示终止的右大括号后必须换行。

```
public void testDownload() {
    System.out.printIn("Hello..");
}
```

- 右大括号后还有else等代码则不换行；

```
if(i==OK) {
    a++;
} else {
    a--;
}
```

- 在 if/else/for/while/do 语句中必须使用大括号。

```
- 正例:
    if(i==OK) {
        a++;
    }
- 反例:
    if(i==OK)
        a++;
```

小括号

- 左小括号和字符之间不允许空格
- 右小括号和字符之间不允许空格

空格符

- 任何二目、三目运算符的左右两边都需要加一个空格。
- if/for/while/switch/do 等保留字与括号之间都必须加空格。
- 任何特俗符号(`/` `,` `)` `:` ...)后有且仅有一个空格。

缩进符

- 采用 4 个空格缩进。

  > 如果使用 tab 缩进，必须设置 1 个 tab 为 4 个空格。
  >
  > IDEA 设置 tab 为 4 个空格时， 请勿勾选 Use tab character；
  >
  > 而在 eclipse 中，必须勾选 insert spaces for tabs。

修饰符

- 修饰符使用顺序应该遵从:

```
annotations
public/protected/private
abstract
static
final
transient
volatile
synchronized
native
strictfp
```

- 缺省的修饰符不可补全。

```
- 接口中方法:
    public final ..
- 接口中常量:
    public static final ..
```

- 类成员与方法访问控制从严:

```
1） 如果不允许外部直接通过 new 来创建对象，那么构造方法必须是 private。
2） 工具类不允许有 public 或 default 构造方法。
3） 类非 static 成员变量并且与子类共享，必须是 protected。
4） 类非 static 成员变量并且仅在本类使用，必须是 private。
5） 类 static 成员变量如果仅在本类使用，必须是 private。
6） 若是 static 成员变量，考虑是否为 final。
7） 类成员方法只供类内部调用，必须是 private。
8） 类成员方法只对继承类公开，那么限制为 protected。
```

代码行

- 单行字符数限制不超过 120 个，超出需要换行，换行时遵循如下原则：

```
1） 第二行相对第一行缩进 4 个空格，从第三行开始，不再继续缩进。
2） 运算符与下文一起换行。
3） 方法调用的点符号与下文一起换行。
4） 方法调用中的多个参数需要换行时，在逗号后进行。
```

- 单个方法的总行数不超过 80 行。

> 说明：包括方法签名、结束右大括号、方法内代码、注释、空行、回车及任何不可见字符的总 行数不超过 80 行。

- 不同逻辑、不同语义、不同业务的代码之间插入一个空行分隔开来以提升可读性。

> 说明：任何情形，没有必要插入多个空行进行隔开。

1.2.3 逻辑规范

封装逻辑

- 不能使用过时的类或方法。
- Object的equals方法容易抛空指针异常，应使用常量或确定有值的对象来调用equals。
- 所有的相同类型的包装类对象之间值的比较，全部使用equals方法比较。

判断逻辑

- 尽量避免if/else结构。

> 说明：如果非得使用if()...else if()...else...方式表达逻辑, 请勿超过3层。

```
if (condition) {
...
return obj;
}
// 接着写else的业务逻辑代码;
```

- 判断条件中应该执行其他复杂的语句 (get/set/is除外)。
- 判断条件复杂时使用括号分隔表达式提高代码可读性。

```
- 正例: if ((i >= j) && (i >= 0))
- 反例: if (i >= j && i >= 0)
```

异常逻辑

- Java 类库中定义的可以通过预检查方式规避的 RuntimeException 异常不应该通过 catch 的方式来处理。

> 比如：NullPointerException，IndexOutOfBoundsException 等等可以代码规避。

- 异常不要用来做流程控制，条件控制。

> 异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式 要低很多。

- 异常捕捉应该尽可能详尽, 不应该大范围捕捉。

> 用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于 简单，在程序上作出分门别类的判断，并提示给用户。

- 不要在 finally 块中使用 return。

> finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句。

1.2.4 声明规范

属性声明

- 定义金额等浮点型的属性不能使用double

> 建议使用int类型代替金额的存储, 单位 `分`。

- 不同类型属性的定义应该单独一行。

```
- 正例:
    int a,b;
    long s;
- 反例:
    int a,b; long s;
```

数组声明

- 数组声明[]放在type后面而不是变量后面。

```
- 正例: private String[] str1, str2, str3;
- 反例: private String str1[], str2[], str3[];
```

1.2.5 注释规范

方法外

- 类、类属性、类方法的注释必须使用 Javadoc 规范，使用 `/** 内容 **/` 格式，不得使用 `// xxx` 方式
- 所有的类和重要方法上都必须添加创建者和创建日期。

```
/**
 * 职责描述
 * ...
 * @author Timi
 * @date 2020-08-20 23:59
 * ...
 */
public class Order {
    /**
     * 职责描述
     * ...
     * @author Timi
     * @date 2020-08-21 00:01
     * ...
```

```
     */
    public void grabOrder(...){
        ...
    }
}
```

- 所有的抽象方法（包括接口中的方法）必须要用 Javadoc 注释、除了返回值、参数、 异常说明外，还必须指出该方法做什么事情，实现什么功能。

方法内

- 方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释 使用`/* 内容 */`注释，注意与代码对齐。
- 所有的枚举类型字段必须要有注释，说明每个数据项的用途。
- 与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持 英文原文即可。

特殊注释

- 特殊注释标记，请注明标记人与标记时间。

> 注意及时处理这些标记，通过标记扫描， 经常清理此类标记。
>
> 线上故障有时候就是来源于这些标记处的代码。

- 待办事宜 (TODO):

> // TODO:（ 标记人，标记时间，[预计处理时间]）
>
> 表示需要实现，但目前还未实现的功能。这实际上是一个 Javadoc 的标签，目前的 Javadoc

- 不能工作(FIXME):

> // FIXME:（标记人，标记时间，[预计处理时间]）
>
> 在注释中用 FIXME 标记某代码是错误的，而且不能工作，需要及时纠正的情况。

- 代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑 等的修改。
- 谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。如果无用，则删除。

1.2.6 并发规范

线程池

- 创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。

```
public class TimerTaskThread extends Thread {
    public TimerTaskThread() {
        super.setName("TimerTaskThread");
        ...
    }
}
```

- 线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。

> 使用线程池的好处是减少在创建和销毁线程上所消耗的时间以及系统资源的开销，解决 资源不足的问题。

> 如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或 者“过度切换”的问题。

- 线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式。

> 这样 的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。
>
> 1）FixedThreadPool 和 SingleThreadPool: 允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。
>
> 2）CachedThreadPool 和 ScheduledThreadPool: 允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

并发锁

- 并发修改同一记录时，避免更新丢失，需要加锁。

> 要么在应用层加锁，要么在缓存加 锁，要么在数据库层使用乐观锁，使用 version 作为更新依据。
>
> 如果每次访问冲突概率小于 20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次 数不得小于 3 次。

- 对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会造 成死锁。

> 线程一需要对表 A、B、C 依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序 也必须是 A、B、C，否则可能出现死锁。

多线程

- 多线程并行处理定时任务，推荐使用 ScheduledExecutorService 。

> Timer 运行多个 TimeTask 时，只要其中之一没有捕获 抛出的异常，其它任务便会自动终止运行。
>
> 使用 ScheduledExecutorService 则没有这个问题。

- 使用 CountDownLatch 进行异步转同步操作，每个线程退出前必须调用 countDown 方法。

> 线程执行代码注意 catch 异常，确保 countDown 方法被执行到，避免主线程无法执行 至 await 方法，直到超时才返回结果。

- ThreadLocal 无法解决共享对象的更新问题，ThreadLocal 对象建议使用 static 修饰。

> 这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享 此静态变量。
>
> 也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只 要是这个线程内定义的)都可以操控这个变量。

- HashMap 在容量不够进行 resize 时由于高并发可能出现死链，导致 CPU 飙升。

> 在 开发过程中可以使用其它数据结构或加锁来规避此风险, 推荐使用CurrentHashMap代替。

- volatile 解决多线程内存不可见问题注意事项。

> 对于一写多读，是可以解决变量同步问题， 但是如果多写，同样无法解决线程安全问题。

1.2.7 日志规范

使用规范

- 应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API。

  > 使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

  ```
  import org.slf4j.Logger;
  import org.slf4j.LoggerFactory;
  private static final Logger logger = LoggerFactory.getLogger(Abc.class);
  ```

- 日志文件至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。
- 对 trace/debug/info 级别的日志输出，必须使用条件输出形式或者使用占位符的方式。

  > logger.debug("Processing trade with id: " + id + " and symbol: " + symbol);
  >
  > 如果日志级别是 warn，上述日志不会打印，但是会执行字符串拼接操作，如果 symbol 是对象， 会执行 toString()方法。
  >
  > 浪费了系统资源，执行了上述操作，最终日志却没有打印。推荐使用以下两种方式:

  ```
  if (logger.isDebugEnabled()) {
      logger.debug("Processing trade with id: " + id + " and symbol: " +
  symbol);
  }
  logger.debug("Processing trade with id: {} and symbol : {} ", id, symbol);
  ```

- 避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置 additivity=false。

  ```
  <logger name="com.taobao.dubbo.config" additivity="false">
  ```

- 可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适 从。

  > 如非必要，请不要在此场景打出 error 级别，避免频繁报警。
  >
  > 注意日志输出的级别，error 级别只记录系统逻辑出错、异常或者重要的错误信息。

- 异常信息应该包括两类信息：案发现场信息和异常堆栈信息。

  > 如果不处理，那么通过 关键字 throws 往上抛出。

命名规范

- 应用中的扩展日志（如打点、临时监控、访问日志等）命名方式:

  > appName_logType_logName.log。
  >
  > appName: 应用名称, 如crm/oms/ins 等;
  >
  > logType:日志类型，如 stats/monitor/access 等;
  >
  > logName:日志描述。
  >
  > 这种命名的好处： 通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。

- 推荐对日志进行分类，如将错误日志和业务日志以名称区分存放。

> 便于开发人员查看，也便于 通过日志对系统进行及时监控。

- 日志文件应该每天滚动一次，日志多的可以每个小时滚动一次。
- 日志的五种级别（按重要程度从低到高排序）并在文件名中提现级别：
  - DEBUG：记录对调试程序有帮助的信息。
  - INFO：记录程序运行现场，虽然此处并未发送错误，但是对排查其他错误信息具有指导意义。
  - WARN：记录程序运行现场，但是更偏向于表明此处有出现潜在错误的可能。
  - ERROR：记录当前程序发生的错误，需要被关注。但是当前发送的错误，没有影响系统继续运行。
  - FATAL：记录当前程序运行出现严重错误事件，并且将会导致应用程序中断。

1.3 缓存规范(All)

- 新版补充

1.4 安全规范(All)

1.4.1 接口规范

- 给前端使用的接口必须用token校验, 给第三的接口必须签名。
- 需要与前端集成的接口必须生成接口文档, 推荐使用Swagger。
- 在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放的机制:

> 如数量限制、疲劳度控制、验证码校验，避免被滥刷而导致资损。
>
> 说明：如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其 它用户，并造成短信平台资源浪费。

- 系统、账户敏感信息必须加密传送。
- 所有接口均需要做必要的参数效验。

> 推荐使用SpringFramework validation框架进行自动效验处理。
>
> 比如: 业务中必须传递的字段应该使用注解@NotEmpty (String) 或 @NotNull (Object)修饰.

1.4.3 数据安全

- 使用主键作为更新条件, 极力避免根据条件修改数据:
  1. 使用非唯一属性修改数据容易造成大面积脏数据 (建议使用物理主键)

```
<update id="updateByPrimaryKeySelective"
parameterType="cn.huolala.cloud.entity.SystemLogs">
    update account.system_logs
    <set>
        <if test="uid != null">
            UID = #{uid,jdbcType=DECIMAL},
        </if>
        <if test="ip != null">
            IP = #{ip,jdbcType=VARCHAR},
        </if>
        ...
    </set>
    where ID = #{id,jdbcType=DECIMAL}
</update>
```

2. 使用Mybatis动态SQL时, 建议去除null属性的字段操作:

```
<insert id="insertSelective"
parameterType="cn.huolala.cloud.entity.SystemLogs">
    insert into account.system_logs
    <trim prefix="(" suffix=")" suffixOverrides=",">
      <if test="id != null">
        ID,
      </if>
      <if test="uid != null">
        UID,
      </if>
      <if test="ip != null">
        IP,
      </if>
        ...
    </trim>
    <trim prefix="values (" suffix=")" suffixOverrides=",">
      <if test="id != null">
        #{id,jdbcType=DECIMAL},
      </if>
      <if test="uid != null">
        #{uid,jdbcType=DECIMAL},
      </if>
      <if test="ip != null">
        #{ip,jdbcType=VARCHAR},
      </if>
        ...
    </trim>
</insert>
```

- 用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入， 禁止字符串拼接 SQL 访问数据库。

  > 使用Mybatis系列框架作为持久层操作数据库工具时, 动态SQL不得使用`$`作为参数占位符, 一律使用`#`;
  >
  > 非Mybatis系列必须做好参数验证和预编译。

- 用户请求传入的任何参数必须做有效性验证。忽略参数校验可能导致：

- page size 过大导致内存溢出
- 恶意 order by 导致数据库慢查询
- 任意重定向
- SQL 注入
- 反序列化注入
- 正则输入源串拒绝服务 ReDoS

> 说明：Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题。
>
> 但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

1.4.2 用户隐私

- 隶属于用户个人的页面或者功能必须进行权限控制校验。

> 防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信 内容、修改他人的订单。

- 用户敏感数据禁止直接展示，必须对展示数据进行脱敏。

> 中国大陆个人手机号码显示为:1589119，隐藏中间 4 位，防止隐私泄露。

1.4.4 党政风控

- 发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过 滤等风控策略。

1.5 代码管理(All)

1.5.1 工具使用

> 推荐使用Git 进行代码管理。
>
> Git基本操作学习推荐廖雪峰教程。

上传

- 提交到服务器的代码必须完整, 不可丢失文件。
- 提交到服务器的代码必须经过自测可执行, 不可影响其他开发。
- 提交的频率至少 1 天/次; 如果只有一次, 建议晚上下班前。

拉取

- 拉取代码前必须先提交所有的已修改文件。
- 开发新的代码前必须拉取服务器最新的代码。
- 拉取的频率最少 1 天/次; 如果只有一次, 建议早上上班前。

1.5.2 版本管理

形成条件

- 每个版本的形成应该具备一个完整的功能点。

  > 一般不建议将完成到一半的功能形成版本并提交, 如有特殊情况必须在提交版本时声明。

- 每个版本推荐只有一个完整的功能点。

  > 一般不建议将多个功能点参杂到一个版本中, 不利于维护/回滚和异常追溯。

- 每个版本提交时必须附有清晰的版本说明。

  > 强烈建议每个提交动作说明代码差异和特殊情况。

版本提交

- 每个功能应该形成一个版本, 但可以等待需求中其他的功能点一起提交到服务器。

  > 如果整个需求当天不能完成, 则应该在下班前提交。

- 每个版本的提交者应该以真实名称作为主体, 如果计算机名不是真实名称首次提交纠正设置。
- 版本说明的格式参考:

  > 需求版本#更新类型;
  > 内容简述...

  ```
  老带新2.0#日常更新/BUG修复/功能优化
  简化分享流程、调整奖励配置模板(新增实物奖励)。
  ```

1.5.3 分支管理

master

- 这个分支最为稳定，这个分支代表项目处于 **可发布** 的状态。

  > 千万不可在此分支上开发代码, 也不可轻易将其他分支的代码合并上来。

- 合并到master的代码必须经过: 代码审核、线上测试、线上试运行。
- 可派生 **hotfix**

develop (dev)

- 作为开发的分支，平行于 **master** 分支。

  > 不可轻易合并代码上来。

- 合并到develop的代码必须经过: 单元测试、系统测试。
- 可派生 **feature**、**release**

hotfix

- 这个分支主要为修复线上特别紧急的bug准备的。
- 必须从 **master** 分支派生。
- 完成后合并回 **develop** 与 **master** 分支。

release

- 这个分支用来发布新版本/试运行。
- 这个分支上可以做一些非常小的bug修复, 大bug需要在 **featrue** 分支上修改。
- 从 **develop** 分支派生。
- 完成后合并回 **develop** 与 **master** 分支。

feature

- 这种分支和我们程序员日常开发最为密切，称作功能分支。
- 必须从 **develop** 分支派生。
- 完成后合并回 **develop** 分支。

生命周期

- master/develop作为主分支, 诞生后不可删除。

  > 所有版本都将在主分支中留痕, 万万不可清除。

- 分支命名格式: 分支类型-需求版本[-其他描述]:
  - hotfix-oldAndNew2.0 **或** hotfix-oldAndNew2.0 -repairTheReward
  - feature-recharge1.0 **或** feature-recharge1.0 -calls
  - release-recharge1.0 **或** release-recharge1.0 -callsBeta
- 任何合并操作后, 被合并分支都应该删除。

  > 合并后的合并分支上已经有被合并分支的版本了, 无需单独再保留被合并分支浪费资源。

二、测试规范

- 测试人员补充

三、文档规范

- 新版补充

四、设计规范

- UI人员补充

4.1 页面布局

4.2 页面色素

4.3 层级样式

4.4 图片图标

4.5 上传下载

五、流程规范

5.1 需求出版

- 产品人员补充

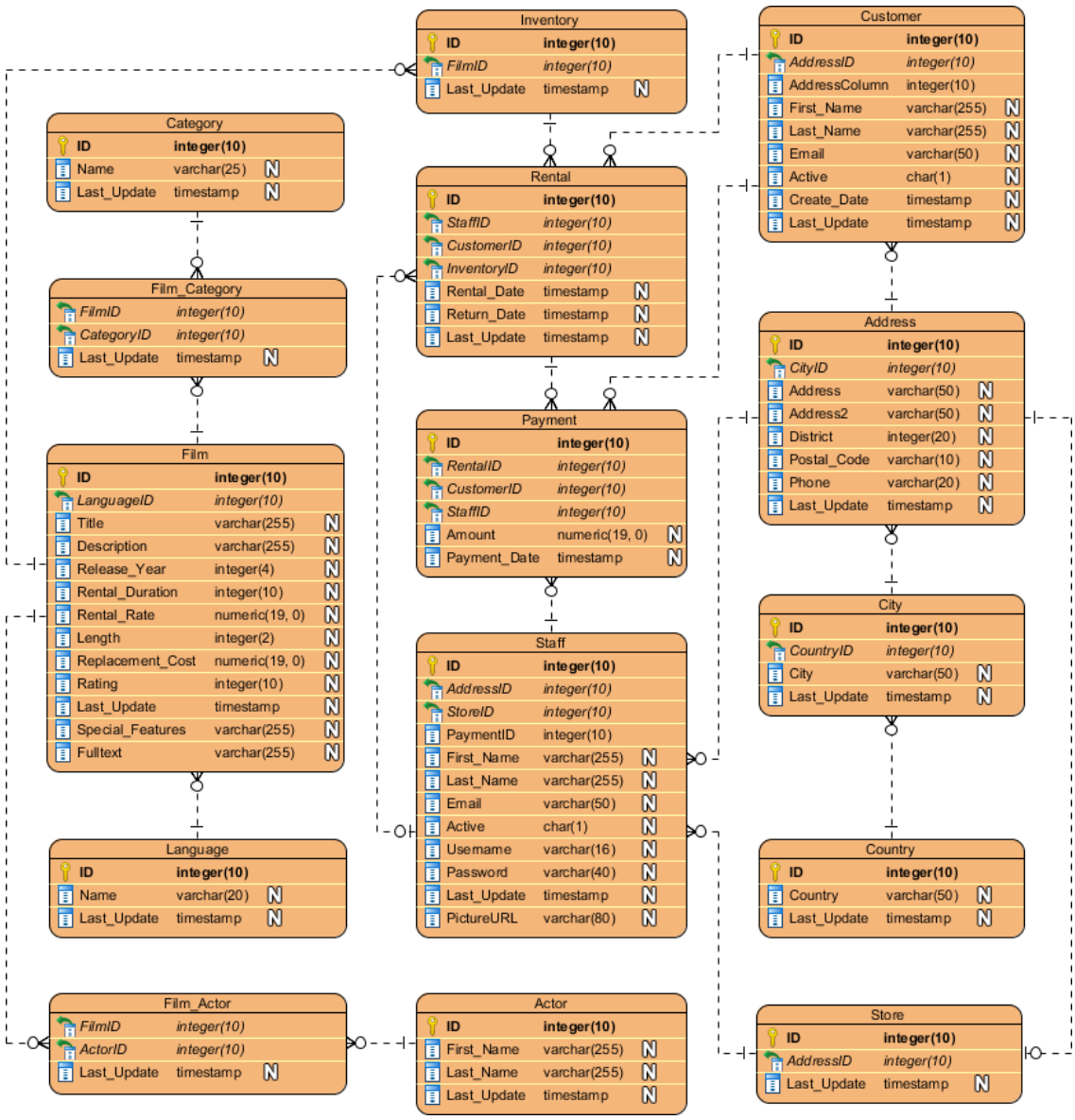
5.2 需求研发

5.2.1 可行性评审

数据模型

> 需求开发前需要先按规范设计数据模型

- 数据模型必须包含数据结构、数据操作、数据约束:
  - 业务对象以及它们之间的关系
  - 关系包含1-1、1-n、n-n、n-1
  - 业务对象命名应该遵从主流的数据库命名规范
  - 字段、字段类型、字段长度、字段用途(注释)
    > 字段注释应该说明: 字段用途,枚举说明等.
  - 字段默认值(如有)、主键(是否自增)、外键(如有)、索引约束(如有)
    > 关系数据库中: 每张表必须至少有一个主键.
- 数据模型建模IDE推荐使用: SqlYog、Navicat、PowerDesigner等



业务流程

> 需求开发前需要先按规范设计业务流程图

- 每个新需求必须设计业务流程图。
  - 开始框，每一流程图只有一个起点, 一般为圆角框。
  - 结束框，流程的中断和结束点, 一般为圆角框。
  - 处理框，表示对事件或结果的处理过程。
  - 决策框，用来根据给定的条件是否满足决定执行某一路径。
  - 流程线，箭头的方向表示流程执行的方向与顺序

    > 两个符号间不得使用双箭头。

  - 连接标识, 用于同一流程图中页和页的连续

    > 或者用于同页内从一个动作框转到另一个动作框

  - 流程标识, 表示在流程图中引用另一个流程。
- 流程图中所用符号应均匀分布, 连线保持合理的长度, 并尽量少用长线。
- 使用各种符号应注意符号的外形和各符号大小的统一。
- 符号内的说明文字尽可能简明。

> 通常按从左向右和从上向下方式书写，并与流向无关。

- 一个大的流程可以由几个小的流程组成。

  > 单个流程过于复杂时，在不影响业务的完整性和连续性的前提下，应拆分为两个及以上子流程。

- 尽量避免流线的交叉。

  > 即使出现流线的交叉，交叉的流线之间也没有任何逻辑关系，并不对流向产生任何影响。

数据流转

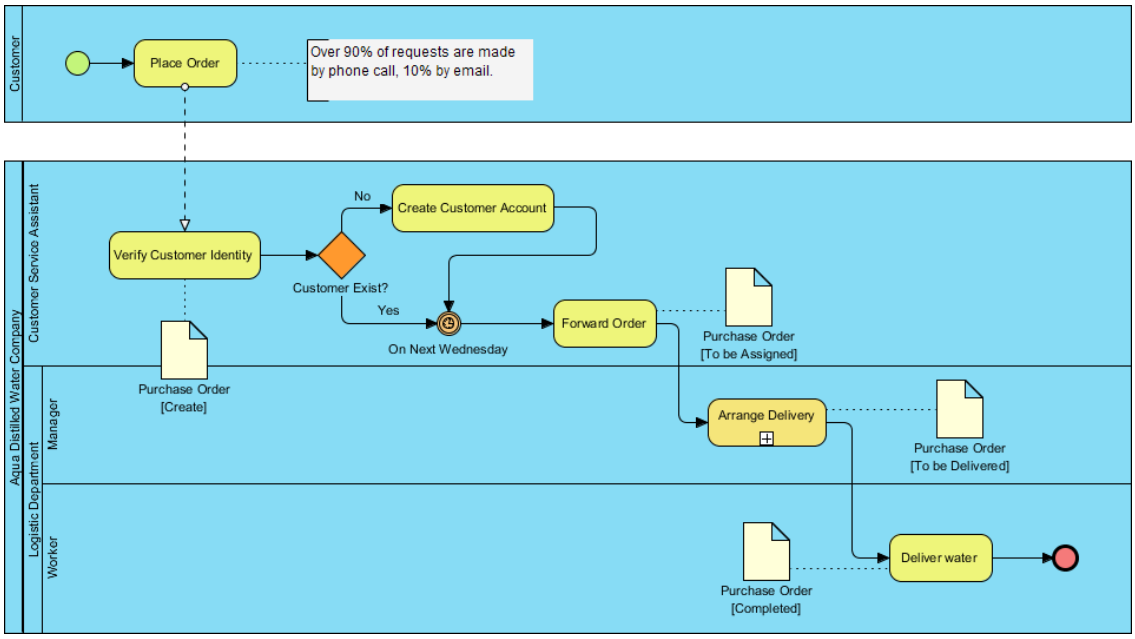
> 需求开发前需要先按规范设计数据流转图 (数据流图)

- 每个系统流程(操作)必须配备数据流转图。
- 每条数据流的起点或者终点必须是加工。

  > 即至少有一端是加工。

- 在分层数据流图中，必须要保持父图与子图平衡。
- 每个加工必须既有输入数据流又有输出数据流。
- 必须要保持数据守恒。

  > 一个加工所有输出数据流中的数据必须能从该加工的输入数据流中直接获得，
  >
  > 或者说是通过该加工能产生的数据。



5.3 需求发版

- 新版补充

5.4 立项评审

- 新版补充

5.5 项目管理

- 新版补充

六、发版规范

- 新版补充

七、部署规范

7.1 服务器规范

- 新版补充

7.2 微服务规范

7.2.1 服务治理

- 分布式场景下拆分的服务必须注册到注册中心。
- 包含核心业务的服务和水平拆分的服务必须集群部署 (2个以上节点)。
- 服务间的调用必须通过注册中心的发现机制进行通讯。