Java开发规范

命名

【规范】类名使用UpperCamelCase 风格，必须遵从驼峰形式，但以下情形例外： （ 领域模型的相关命名 ）DO / BO / DTO / VO 等。

正例： MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

反例： macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

【规范】方法名、参数名、成员变量、局部变量都统一使用lowerCamelCase 风格，必须遵从驼峰形式。

正例： localValue / getHttpMessage() / inputUserId

【规范】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。

【规范】抽象类命名使用 Abstract 或 Base 开头 ； 异常类命名使用 Exception 结尾 ； 测试类命名以它要测试的类的名称开始，以 Test 结尾。枚举类名建议带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。

【规范】POJO 类中布尔类型的变量，都不要加 is ，否则部分框架解析会引起序列化错误。

【规范】各层命名规约：

A) Service / DAO 层方法命名规约

1 ） 获取单个对象的方法用 get 做前缀。

2 ） 获取多个对象的方法用 list 做前缀（习惯：getXXXList）。

3 ） 获取统计值的方法用 count 做前缀。

4 ） 插入的方法用 save（ 推荐 ） 或 insert 做前缀。

5 ） 删除的方法用 remove（ 推荐 ） 或 delete 做前缀。

6 ） 修改的方法用 update 做前缀(或modify)。

B) 领域模型命名规约

1 ） 数据对象： xxxDO ， xxx 即为数据表名。

2 ） 数据传输对象： xxxDTO ， xxx 为业务领域相关的名称。

3 ） 展示对象： xxxVO ， xxx 一般为网页名称。

4 ） POJO 是 DO / DTO / BO / VO 的统称，禁止命名成 xxxPOJO 。

常量

【规范】不允许任何魔法值（ 即未经定义的常量 ） 直接出现在代码中。

反例： String key =" Id # taobao \_"+ tradeId；

cache . put(key , value);

格式规约

【风格】单行太长需换行

【风格】方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

OOP规约

【效率】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

【规范】所有的覆写方法，必须加@ Override 注解。

【规范】对外暴露的接口签名，原则上不允许修改方法签名，避免对接口调用方产生影响。接口过时必须加@Deprecated 注解，并清晰地说明采用的新接口或者新服务是什么。

【规范】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。

正例： " test " .equals(object);

反例： object.equals( " test " );

【规范】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。(注意空指针)

说明：对于 Integer var =?在-128 至 127 之间的赋值， Integer 对象是在IntegerCache . cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用==进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

【规范】关于基本数据类型与包装数据类型的使用标准如下：

1 ） 所有的 POJO 类属性必须使用包装数据类型。

2 ） RPC 方法的返回值和参数必须使用包装数据类型。

3 ） 所有的局部变量【推荐】使用基本数据类型。

【强制】序列化类新增属性时，请不要修改 serialVersionUID 字段，避免反序列失败 ； 如果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值。

【规范】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在 init 方法中。

【规范】使用索引访问用 String 的 split 方法得到的数组时，需做最后一个分隔符后有无内容的检查，否则会有抛 IndexOutOfBoundsException 的风险。

说明：

String str = "a,b,c,,";

String[] ary = str.split(",");

//预期大于 3，结果是 3

System.out.println(ary.length);

【规范】当一个类有多个构造方法，或者多个同名方法，这些方法应该按顺序放置在一起，便于阅读。

【风格】类内方法定义顺序依次是：公有方法或保护方法 > 私有方法 > getter / setter方法。

【效率】final 可提高程序响应效率，声明成 final 的情况：

1 ） 不需要重新赋值的变量，包括类属性、局部变量。

2 ） 对象参数前加 final ，表示不允许修改引用的指向。

3 ） 类方法确定不允许被重写。

4 ）例子：final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);

集合处理

【强制】关于 hashCode 和 equals 的处理，遵循如下规则：

1） 只要重写 equals ，就必须重写 hashCode 。

2） 因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须重写这两个方法。

3） 如果自定义对象做为 Map 的键，那么必须重写 hashCode 和 equals 。

【强制】不要在 foreach 循环里进行元素的 remove / add 操作。 remove 元素请使用 Iterator方式，如果并发操作，需要对 Iterator 对象加锁。

反例：

List<String> a = new ArrayList<String>();

a.add("1");

a.add("2");

for (String temp : a) {

if("1".equals(temp)){

a.remove(temp);

}

}

说明：以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的结果吗？(java.util.ConcurrentModificationException)

正例：

Iterator<String> it = a.iterator();

while(it.hasNext()){

String temp = it.next();

if(删除元素的条件){

it.remove();

}

}

【规范】集合初始化时，尽量指定集合初始值大小。

说明： ArrayList 尽量使用 ArrayList(int initialCapacity) 初始化。

【规范】使用 entrySet 遍历 Map 类集合 KV，而不是 keySet 方式进行遍历。

说明:keySet 其实是遍历了 2 次，一次是转为 Iterator 对象，另一次是从 hashMap 中取出 key 所对应的 value。而 entrySet 只是遍历了一次就把 key 和 value 都放到了 entry 中，效 率更高。如果是 JDK8，使用 Map.foreach 方法。

Map<String, String> map = new HashMap<String, String>();

map.put("1", "@@");

map.put("2", "##");

/\*\*

\* JDK8推荐使用

\*/

map.forEach((K, V) -> {

System.out.println("Key : " + K);

System.out.println("Value : " + V);

});

/\*\*

\* foreach推荐使用

\*/

for (Map.Entry<String, String> entry : map.entrySet()) {

System.out.println("Key : " + entry.getKey());

System.out.println("Value : " + entry.getValue());

}

/\*\*

\* 不推荐使用

\*/

for (String key : map.keySet()) {

System.out.println("Key : " + key);

System.out.println("Value : " + map.get(key));

}

【强制】高度注意 Map 类集合 K/V 能不能存储 null 值的情况，如下表格:

集合类 Key Value Super 说明

Hashtable 不允许为 null 不允许为 null Dictionary 线程安全

ConcurrentHashMap 不允许为 null 不允许为 null AbstractMap 分段锁技术

TreeMap 不允许为 null 允许为 null AbstractMap 线程不安全

HashMap 允许为 null 允许为 null AbstractMap 线程不安全

并发处理

【规范】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。

说明:资源驱动类、工具类、单例工厂类都需要注意。

【规范】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。

正例:

public class TimerTaskThread extends Thread { public TimerTaskThread(){

super.setName("TimerTaskThread"); ... }

【规范】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。

说明:使用线程池的好处是减少在创建和销毁线程上所花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。

【规范】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。 说明:Executors 返回的线程池对象的弊端如下:

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。

【效率】高并发时，同步调用应该去考量锁的性能损耗。能用无锁数据结构，就不要用锁;能 锁区块，就不要锁整个方法体;能用对象锁，就不要用类锁。

【强制】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。

说明: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。

【规范】并发修改同一记录时，避免更新丢失，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。

说明: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。

【规范】多线程并行处理定时任务时，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。

【规范】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 开发过程中注意规避此风险。

控制语句

【规范】在一个 switch 块内，每个 case 要么通过 break/return 等来终止，要么注释说明程 序将继续执行到哪一个 case 为止;在一个 switch 块内，都必须包含一个 default 语句并且 放在最后，即使它什么代码也没有。

【规范】在 if/else/for/while/do 语句中必须使用大括号，即使只有一行代码，避免使用 下面的形式:if (condition) statements;

【规范】推荐尽量少用 else， if-else 的方式可以改写成:

if(condition){

...

return obj; }

// 接着写 else 的业务逻辑代码;

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

超过请使用状态设计模式 或者 卫语句。

卫语句示例：

public void today() { if (isBusy()) {

System.out.println(“change time.”); return;

}

if (isFree()) {

System.out.println(“go to travel.”);

return;

}

System.out.println(“stay at home to learn Alibaba Java Coding Guidelines.”);

return;

}

【规范】除常用方法(如 getXxx/isXxx)等外，不要在条件判断中执行其它复杂的语句，将复 杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明:很多 if 语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么 样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢?

正例:

//伪代码如下

boolean existed = (file.open(fileName, "w") != null) && (...) || (...); if (existed) {

... }

反例:

if ((file.open(fileName, "w") != null) && (...) || (...)) { ...

}

【规范】方法中需要进行参数校验的场景:

1) 调用频次低的方法。

2) 执行时间开销很大的方法，参数校验时间几乎可以忽略不计，但如果因为参数错误导致

中间执行回退，或者错误，那得不偿失。

3) 需要极高稳定性和可用性的方法。

4) 对外提供的开放接口，不管是RPC/API/HTTP接口。

5) 敏感权限入口。

【规范】方法中不需要参数校验的场景:

1) 极有可能被循环调用的方法，不建议对参数进行校验。但在方法说明里必须注明外部参

数检查。

2) 底层的方法调用频度都比较高，一般不校验。毕竟是像纯净水过滤的最后一道，参数错误不太可能到底层才会暴露问题。一般 DAO 层与 Service 层都在同一个应用中，部署在同一 台服务器中，所以 DAO 的参数校验，可以省略。

3) 被声明成private只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参 数已经做过检查或者肯定不会有问题，此时可以不校验参数。

注释规约

【规范】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。

【规范】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。

说明:对子类的实现要求，或者调用注意事项，请一并说明。

【风格】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/\* \*/注释，注意与代码对齐。

【规范】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。

【规范】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。

【规范】注释掉的代码尽量要配合说明，而不是简单的注释掉。

说明:代码被注释掉有两种可能性:

1)后续会恢复此段代码逻辑。

2)永久不用。前者如果没 有备注信息，难以知晓注释动机。

后者建议直接删掉(代码仓库保存了历史代码)。

【风格】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。

1) 待办事宜(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

还没有实现，但已经被广泛使用。只能应用于类，接口和方法(因为它是一个 Javadoc 标签)。

2) 错误，不能工作(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。

异常

【规范】异常不要用来做流程控制，条件控制，因为异常的处理效率比条件分支低。

【规范】对大段代码进行 try-catch，这是不负责任的表现。catch 时请分清稳定代码和非稳 定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的 catch 尽可能进行区分 异常类型，再做对应的异常处理。

【规范】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请 将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的 内容。

【强制】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回 滚事务。

【规范】不能在 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。

【规范】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分 说明什么情况下会返回 null 值。调用方需要进行 null 判断防止 NPE 问题。

【规范】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景:

1) 返回类型为包装数据类型，有可能是null，返回int值时注意判空。

反例:public int f(){ return Integer 对象}; 如果为 null，自动解箱抛 NPE。

2) 数据库的查询结果可能为null。

3) 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。

4) 远程调用返回对象，一律要求进行NPE判断。

5) 对于Session中获取的数据，建议NPE检查，避免空指针。

6) 级联调用obj.getA().getB().getC();一连串调用，易产生NPE。

【规范】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api 开放接口必须 使用“错误码”;而应用内部推荐异常抛出;跨应用间 RPC 调用优先考虑使用 Result 方式，封 装 isSuccess、“错误码”、“错误简短信息”。

说明:关于 RPC 方法返回方式使用 Result 方式的理由:

1)使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。

2)如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用 端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输 的性能损耗也是问题。

【规范】避免出现重复的代码(Don’t Repeat Yourself)，即DRY原则。

日志

【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);

【规范】日志文件推荐至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。

【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。

logType:日志类型，推荐分类有 stats/desc/monitor/visit 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。

正例:mppserver 应用中单独监控时区转换异常，如: mppserver\_monitor\_timeZoneConvert.log

说明:推荐对日志进行分类，错误日志和业务日志尽量分开存放，便于开发人员查看，也便于 通过日志对系统进行及时监控。

【规范】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。

说明:logger.debug("Processing trade with id: " + id + " 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。 正例:(条件)

if (logger.isDebugEnabled()) {

logger.debug("Processing trade with id: " + id + " symbol: " + symbol);

}

正例:(占位符)

logger.debug("Processing trade with id: {} 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。

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

【规范】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。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必 要，请不要在此场景打出 error 级别。

【规范】谨慎地记录日志。生产环境禁止输出 debug 日志;有选择地输出 info 日志;如果使 用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘 撑爆，并记得及时删除这些观察日志。

说明:大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请

思考:这些日志真的有人看吗?看到这条日志你能做什么?能不能给问题排查带来好处?

其它

【效率】在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。 说明:不要在方法体内定义:Pattern pattern = Pattern.compile(规则);

【规范】获取当前毫秒数 System.currentTimeMillis(); 而不是 new Date().getTime();

说明:如果想获取更加精确的纳秒级时间值，用 System.nanoTime()。在 JDK8 中，针对统计 时间等场景，推荐使用Instant 类。

【规范】对于“明确停止使用的代码和配置”，如方法、变量、类、配置文件、动态配置属性等要坚决从程序中清理出去，避免造成过多垃圾。

单元测试

【强制】好的单元测试必须遵守 AIR 原则。

说明:单元测试在线上运行时，感觉像空气(AIR)一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。

 A:Automatic(自动化)

 I:Independent(独立性)

 R:Repeatable(可重复)

【强制】单元测试应该是全自动执行的，并且非交互式的。测试框架通常是定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测 试中不准使用 System.out 来进行人肉验证，必须使用 assert 来验证。

【强制】保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间 决不能互相调用，也不能依赖执行的先后次序。

反例:method2 需要依赖 method1 的执行，将执行结果做为 method2 的输入。

【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。

说明:只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的 交互逻辑，那是集成测试的领域。

【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。

说明:新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。

【推荐】单元测试的基本目标:语句覆盖率达到 70%;核心模块的语句覆盖率和分支覆盖率都 要达到 100%

说明:在工程规约的应用分层中提到的 DAO 层，Manager 层，可重用度高的 Service，都应该 进行单元测试。

【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。

 B:Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。

C:Correct，正确的输入，并得到预期的结果。

 D:Design，与设计文档相结合，来编写单元测试。

 E:Error，强制错误信息输入(如:非法数据、异常流程、非业务允许输入等)，并得 到预期的结果。

【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者 对单元测试产生的数据有明确的前后缀标识。

正例:在 RDC 内部单元测试中，使用 RDC\_UNIT\_TEST\_的前缀标识数据。

【推荐】在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好 覆盖所有测试用例(UC)。

【推荐】单元测试作为一种质量保障手段，不建议项目发布后补充单元测试用例，建议在项 目提测前完成单元测试。

【参考】不要对单元测试存在如下误解:

 那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。

单元测试代码是多余的。汽车的整体功能与各单元部件的测试正常与否是强相关的。 

单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。

 单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障。

MySQL开发规范

建表规约

【规范】表达是与否概念的字段，必须使用 is\_xxx 的方式命名，数据类型是 unsigned tinyint ( 1表示是，0表示否)，此规则同样适用于odps建表。 说明:任何字段如果为非负数，必须是 unsigned。

【规范】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。

正例:getter\_admin，task\_config，level3\_name

反例:GetterAdmin，taskConfig，level\_3\_name

【规范】唯一索引名为 uk\_字段名;普通索引名则为 idx\_字段名。

【规范】小数类型为 decimal，禁止使用 float 和 double。

说明: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。

【规范】如果存储的字符串长度几乎相等，使用 char 定长字符串类型。

【效率】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。

【规范】表的命名最好是加上“业务名称\_表的作用”。

正例:tiger\_task / tiger\_reader / mpp\_config

【规范】库名与应用名称尽量一致。

【规范】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。

【效率】字段允许适当冗余，以提高性能，但是必须考虑数据同步的情况。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。 正例:商品类目名称使用频率高，字段长度短，名称基本一成不变，可在相关联的表中冗余存 储类目名称，避免关联查询。

【效率】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。 说明:如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表。

【效率】合适的字符存储长度，不但节约数据库表空间、节约索引存储，更重要的是提升检 索速度。

正例:人的年龄用 unsigned tinyint(表示范围 0-255，人的寿命不会超过 255 岁);海龟 就必须是 smallint，但如果是太阳的年龄，就必须是 int;如果是所有恒星的年龄都加起来， 那么就必须使用 bigint。

索引规约

【效率】业务上具有唯一特性的字段，即使是组合字段，也必须建成唯一索引。

说明:不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明 显的;另外，即使在应用层做了非常完善的校验和控制，只要没有唯一索引，根据墨菲定律， 必然有脏数据产生。

【规范】超过三个表禁止 join。需要 join 的字段，数据类型保持绝对一致;多表关联查询 时，保证被关联的字段需要有索引。

说明:即使双表 join 也要注意表索引、SQL 性能。

【规范】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据 实际文本区分度决定索引长度。

说明:索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分 度会高达 90%以上，可以使用 count(distinct left(列名, 索引长度))/count(\*)的区分度 来确定。

【规范】页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。

说明:索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索 引。

【规范】如果有 order by 的场景，请注意利用索引的有序性。order by 最后的字段是组合 索引的一部分，并且放在索引组合顺序的最后，避免出现 file\_sort 的情况，影响查询性能。

正例:where a=? and b=? order by c; 索引:a\_b\_c

反例:索引中有范围查找，那么索引有序性无法利用，如:WHERE a>10 ORDER BY b; 索引 a\_b 无法排序。

【效率】利用覆盖索引来进行查询操作，来避免回表操作。

说明:如果一本书需要知道第 11 章是什么标题，会翻开第 11 章对应的那一页吗?目录浏览 一下就好，这个目录就是起到覆盖索引的作用。 正例:能够建立索引的种类:主键索引、唯一索引、普通索引，而覆盖索引是一种查询的一种 效果，用explain的结果，extra列会出现:using index。

【效率】利用延迟关联或者子查询优化超多分页场景。

说明:MySQL 并不是跳过 offset 行，而是取 offset+N 行，然后返回放弃前 offset 行，返回 N 行，那当 offset 特别大的时候，效率就非常的低下，要么控制返回的总页数，要么对超过 特定阈值的页数进行 SQL 改写。

正例:先快速定位需要获取的 id 段，然后再关联:（优化在可以少查表1的很多字段）

SELECT a.\* FROM 表 1 a, (select id from 表 1 where 条件 LIMIT 100000,20 ) b where a.id=b.id

【效率】SQL 性能优化的目标:至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好。

说明:

1)consts 单表中最多只有一个匹配行(主键或者唯一索引)，在优化阶段即可读取到数据。

2)ref 指的是使用普通的索引(normal index)。

3)range 对索引进行范围检索。

反例:explain 表的结果，type=index，索引物理文件全扫描，速度非常慢，这个 index 级 别比较 range 还低，与全表扫描是小巫见大巫。

【规范】建组合索引的时候，区分度最高的在最左边。

正例:如果 where a=? and b=? ，a 列的几乎接近于唯一值，那么只需要单建 idx\_a 索引即 可。

说明:存在非等号和等号混合判断条件时，在建索引时，请把等号条件的列前置。如:where a>? and b=? 那么即使 a 的区分度更高，也必须把 b 放在索引的最前列。

【说明】创建索引时避免有如下极端误解:

1)误认为一个查询就需要建一个索引。

2)误认为索引会消耗空间、严重拖慢更新和新增速度。

3)误认为唯一索引一律需要在应用层通过“先查后插”方式解决。

SQL规约

【规范】不要使用 count(列名)或 count(常量)来替代 count(\*)，count(\*)就是 SQL92 定义 的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明:count(\*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

【说明】count(distinct col) 计算该列除 NULL 之外的不重复数量。注意 count(distinct col1, col2) 如果其中一列全为NULL，那么即使另一列有不同的值，也返回为0。

【说明】当某一列的值全是 NULL 时，count(col)的返回结果为 0，但 sum(col)的返回结果为 NULL，因此使用 sum()时需注意 NPE 问题。

正例:可以使用如下方式来避免sum的NPE问题:SELECT IF(ISNULL(SUM(g)),0,SUM(g)) FROM table;

【说明】使用 ISNULL()来判断是否为 NULL 值。注意:NULL 与任何值的直接比较都为 NULL。

说明:

1) NULL<>NULL的返回结果是NULL，而不是false。

2) NULL=NULL的返回结果是NULL，而不是true。

3) NULL<>1的返回结果是NULL，而不是true。

【规范】不得使用外键与级联，一切外键概念必须在应用层解决。

说明:(概念解释)学生表中的 student\_id 是主键，那么成绩表中的 student\_id 则为外键。 如果更新学生表中的 student\_id，同时触发成绩表中的 student\_id 更新，则为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群;级联更新是强阻塞，存在数 据库更新风暴的风险;外键影响数据库的插入速度。

【规范】数据订正时，删除和修改记录时，要先 select，避免出现误删除，确认无误才能执行更新语句。

【效率】in 操作能避免则避免，若实在避免不了，需要仔细评估 in 后边的集合元素数量，控制在 1000 个之内。（可以用用 EXISTS ，NOT EXISTS 或 JOIN代替）

【规范】如果有全球化需要，所有的字符存储与表示，均以 utf-8 编码，那么字符计数方法 注意:

说明:

SELECT LENGTH("轻松工作"); 返回为12

SELECT CHARACTER\_LENGTH("轻松工作"); 返回为4 如果要使用表情，那么使用 utfmb4 来进行存储，注意它与 utf-8 编码的区别。

【规范】TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少，但 TRUNCATE无事务且不触发 trigger，有可能造成事故，故不建议在开发代码中使用此语句。

说明:TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同。

ORM规约

【规范】POJO 类的 boolean 属性不能加 is，而数据库字段必须加 is\_，要求在 resultMap 中 进行字段与属性之间的映射。

说明:参见定义 POJO 类以及数据库字段定义规定，在 sql.xml 增加映射，是必须的。

【安全】配置XML文件时注意SQL注入问题。

【规范】不允许直接拿 HashMap 与 Hashtable 作为查询结果集的输出。

【强制】更新数据表记录时，必须同时更新记录对应的 gmt\_modified 字段值为当前时间。

【规范】不要写一个大而全的数据更新接口，传入为 POJO 类，不管是不是自己的目标更新字 段，都进行 update table set c1=value1,c2=value2,c3=value3; 这是不对的。执行 SQL 时，尽量不要更新无改动的字段，一是易出错;二是效率低;三是 binlog 增加存储。

【规范】@Transactional 事务不要滥用。事务会影响数据库的 QPS，另外使用事务的地方需 要考虑各方面的回滚方案，包括缓存回滚、搜索引擎回滚、消息补偿、统计修正等。

工程规约

【说明】图中默认上层依赖于下层，箭头关系表示可直接依赖，如:开放接口层可以依赖于Web 层，也可以直接依赖于 Service 层，依此类推。

 开放接口层:可直接封装 Service接口暴露成 RPC 接口;通过 Web 封装成 http 接口;网关控 制层等。

 终端显示层:各个端的模板渲染并执行显示层。当前主要是 velocity 渲染，JS 渲染，JSP 渲 染，移动端展示层等。

 Web 层:主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。(Controller)

 Service 层:相对具体的业务逻辑服务层。

 Manager 层:通用业务处理层，它有如下特征:

1) 对第三方平台封装的层，预处理返回结果及转化异常信息;

2) 对Service层通用能力的下沉，如缓存方案、中间件通用处理;

3) 与DAO层交互，对DAO的业务通用能力的封装。

 DAO 层:数据访问层，与底层 MySQL、Oracle、Hbase 进行数据交互。

 外部接口或第三方平台:包括其它部门 RPC 开放接口，基础平台，其它公司的 HTTP 接口。

【规范】(分层异常处理规约)在 DAO 层，产生的异常类型有很多，无法用细粒度的异常进 行catch，使用catch(Exception e)方式，并throw new DAOException(e)，不需要打印日志，因为日志在 Manager/Service 层一定需要捕获并打到日志文件中去，如果同台服务器 再打日志，浪费性能和存储。在 Service 层出现异常时，必须记录出错日志到磁盘，尽可能带 上参数信息，相当于保护案发现场。如果 Manager 层与 Service 同机部署，日志方式与 DAO 层处理一致，如果是单独部署，则采用与 Service 一致的处理方式。Web 层绝不应该继续往上抛异常，因为已经处于顶层，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面，加上用户容易理解的错误提示信息。开放接口层要将异常处理成错误码 和错误信息方式返回。

【规范】分层领域模型规约:

 DO(Data Object):与数据库表结构一一对应，通过 DAO 层向上传输数据源对象。(Entity)

 DTO(Data Transfer Object):数据传输对象，Service 和 Manager 向外传输的对象。

 BO(Business Object):业务对象。可以由 Service 层输出的封装业务逻辑的对象。

 QUERY:数据查询对象，各层接收上层的查询请求。注:超过 2 个参数的查询封装，禁止 使用 Map 类来传输。

 VO(View Object):显示层对象，通常是 Web 向模板渲染引擎层传输的对象。

服务器规约

【效率】高并发服务器建议调小 TCP 协议的 time\_wait 超时时间。

说明:操作系统默认 240 秒后，才会关闭处于 time\_wait 状态的连接，在高并发访问下，服务器端会因为处于 time\_wait 的连接数太多，可能无法建立新的连接，所以需要在服务器上 调小此等待值。

正例:在 linux 服务器上请通过变更/etc/sysctl.conf 文件去修改该缺省值(秒):

net.ipv4.tcp\_fin\_timeout = 30

【效率】调大服务器所支持的最大文件句柄数(File Descriptor，简写为fd)。

说明:主流操作系统的设计是将 TCP/UDP 连接采用与文件一样的方式去管理，即一个连接对 应于一个 fd。主流的 linux 服务器默认所支持最大 fd 数量为 1024，当并发连接数很大时很 容易因为 fd 不足而出现“open too many files”错误，导致新的连接无法建立。 建议将 linux 服务器所支持的最大句柄数调高数倍(与服务器的内存数量相关)。

【规范】给 JVM 设置-XX:+HeapDumpOnOutOfMemoryError 参数，让 JVM 碰到 OOM 场景时输出 dump 信息。

说明:OOM 的发生是有概率的，甚至有规律地相隔数月才出现一例，出现时的现场信息对查错 非常有价值。

【规范】服务器内部重定向使用 forward;外部重定向地址使用 URL 拼装工具类来生成，否则 会带来 URL 维护不一致的问题和潜在的安全风险。

安全规约

【安全】隶属于用户个人的页面或者功能必须进行权限控制校验。

说明:防止没有做水平权限校验就可随意访问、操作别人的数据，比如查看、修改别人的订单。

【安全】用户敏感数据禁止直接展示，必须对展示数据脱敏。

说明:查看个人手机号码会显示成:158\*\*\*\*9119，隐藏中间 4 位，防止隐私泄露。

【安全】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。

【安全】用户请求传入的任何参数必须做有效性验证。

说明: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效果。

【安全】禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。

【安全】表单、AJAX 提交必须执行 CSRF 安全过滤。

说明:CSRF(Cross-site request forgery)跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户 不知情情况下对数据库中用户参数进行相应修改。

【安全】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限制， 如数量限制、疲劳度控制、验证码校验，避免被滥刷、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。

【安全】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。