目录

[Java规范 3](#_Toc478819403)

[一、 编程规范 3](#_Toc478819404)

[1.1 命名规范 3](#_Toc478819405)

[1.1.1 参数类型String[] args 3](#_Toc478819406)

[1.1.2 常量命名下划线 3](#_Toc478819407)

[1.1.3 特殊类名前后缀 3](#_Toc478819408)

[1.1.4 布尔变量不加is前缀（POJO类中） 3](#_Toc478819409)

[1.1.5 包名单数 3](#_Toc478819410)

[1.1.6 接口方法&属性定义不加修饰符 3](#_Toc478819411)

[1.1.7 枚举名称定义 4](#_Toc478819412)

[1.1.8 各层命名规约 4](#_Toc478819413)

[1.2 常量定义 4](#_Toc478819414)

[1.2.1 禁止魔法值 4](#_Toc478819415)

[1.2.2 Long型赋值 4](#_Toc478819416)

[1.3 格式规约 4](#_Toc478819417)

[1.3.1 保留字括号空格 4](#_Toc478819418)

[1.3.2 换行 4](#_Toc478819419)

[1.3.3 方法定义传参 5](#_Toc478819420)

[1.3.4 IDE文件换行符 5](#_Toc478819421)

[1.4 OOP规约 6](#_Toc478819422)

[1.4.1 访问静态变量/静态方法 6](#_Toc478819423)

[1.4.2 @Override 6](#_Toc478819424)

[1.4.3 接口签名修改 6](#_Toc478819425)

[1.4.4 值比较 6](#_Toc478819426)

[1.4.5 基本数据类型 vs 包装数据类型 6](#_Toc478819427)

[1.4.6 POJO不设属性默认值 6](#_Toc478819428)

[1.4.7 序列化 6](#_Toc478819429)

[1.4.8 业务逻辑初始化 7](#_Toc478819430)

[1.4.9 POJO toString()方法 7](#_Toc478819431)

[1.5 集合处理 7](#_Toc478819432)

[1.5.1 Map / Set key值重写hashCode和equals 7](#_Toc478819433)

[1.5.2 ArrayList vs subList 7](#_Toc478819434)

[1.5.3 sublist 7](#_Toc478819435)

[1.5.4 集合转数组 toArray(T[] array) 7](#_Toc478819436)

[1.5.5 集合通配符 8](#_Toc478819437)

[1.5.6 集合初始化大小 9](#_Toc478819438)

[1.5.7 K/V存储null 9](#_Toc478819439)

[1.6 并发处理 9](#_Toc478819440)

[1.6.1 单例线程安全 9](#_Toc478819441)

[1.6.2 线程由线程池提供 9](#_Toc478819442)

[1.6.3 日期类SimpleDateFormat & DateUtils 10](#_Toc478819443)

[1.6.4 锁的使用 10](#_Toc478819444)

[1.6.5 并发修改 10](#_Toc478819445)

[1.6.6 多线程定时任务 10](#_Toc478819446)

[1.6.7 不使用Executors，用ThreadPoolExecutor创建线程池 10](#_Toc478819447)

[1.6.8 线程/线程池创建时指定有意义的名称 10](#_Toc478819448)

[1.6.9 CountDownLatch 必须执行 countDown方法退出 11](#_Toc478819449)

[1.6.10 双重检查锁 volatile 11](#_Toc478819450)

[1.6.11 【参考】注意HashMap的扩容死链，导致CPU飙升的问题。 11](#_Toc478819451)

[1.6.12 ThreadLocal无法解决共享对象的更新 12](#_Toc478819452)

[1.7 控制语句 12](#_Toc478819453)

[1.7.1 Switch 12](#_Toc478819454)

[1.7.2 If else 判断 12](#_Toc478819455)

[1.7.3 参数校验 12](#_Toc478819456)

[1.8 注释规约 13](#_Toc478819457)

[1.8.1 Javadoc注释 13](#_Toc478819458)

[1.8.2 单行注释 // 13](#_Toc478819459)

[1.8.3 枚举 13](#_Toc478819460)

[1.9 其他 13](#_Toc478819461)

[1.9.1 预编译正则表达式 13](#_Toc478819462)

[1.9.2 Bean 属性copy 13](#_Toc478819463)

[1.9.3 获取时间 13](#_Toc478819464)

[二、 异常日志 14](#_Toc478819465)

[2.1 异常处理 14](#_Toc478819467)

[2.1.1 不捕获RuntimeException子类 14](#_Toc478819468)

[2.1.2 Try catch 14](#_Toc478819469)

[2.1.3 Finally return 14](#_Toc478819470)

[2.1.4 Null 值返回 15](#_Toc478819471)

[2.1.5 抛异常 vs 返回错误码 15](#_Toc478819472)

[2.2 日志规约 15](#_Toc478819473)

[2.2.1 日志保存日期 15](#_Toc478819474)

[2.2.2 异常信息 15](#_Toc478819475)

[三、 MYSQL规约 15](#_Toc478819476)

[3.1 建表规约 15](#_Toc478819478)

[3.1.1 是否字段 0/1 15](#_Toc478819479)

[3.1.2 小数类型decimal 15](#_Toc478819480)

[3.1.3 Text:字段长短超过5000 15](#_Toc478819481)

[3.2 索引规约 16](#_Toc478819482)

[3.2.1 唯一索引 16](#_Toc478819483)

[3.2.2 Varchar字段索引长度 16](#_Toc478819484)

[3.2.3 索引排序 16](#_Toc478819485)

[3.3 SQL规约 16](#_Toc478819486)

[3.3.1 Count 16](#_Toc478819487)

[3.3.2 不使用存储过程 16](#_Toc478819488)

[3.3.3 TRUNCATE vs DELETE 16](#_Toc478819489)

[3.3.4 Xml配置参数 16](#_Toc478819490)

[3.3.5 动态值判断 16](#_Toc478819491)

Java规范

# 编程规范

## 命名规范

### 参数类型String[] args

推荐使用String[] args ,明确声明了参数类型为String[]，而String args[] 只声明为String

### 常量命名下划线

单词用下划线隔开

正例： MAX\_STOCK\_COUNT

### 特殊类名前后缀

抽象类（不是接口）命名用Abstract/ Base开头，异常类名以Exception结尾

AbstractUserService BaseUserService

BusinessException

### 布尔变量不加is前缀（POJO类中）

Reason：部分框架解析错误

反例：boolean isSuccess

定义为基本数据类型boolean isSuccess；的属性，它的方法也是isSuccess()，RPC框架在反向解析的时候，“以为”对应的属性名称是success，导致属性获取不到，进而抛出异常。

### 包名单数

统一使用单数形式，类名才可以复数

### 接口方法&属性定义不加修饰符

正例：void f();

基础变量：String COMPANY = "alibaba";

反例：接口方法定义：public abstract void f();

接口与变量不加修饰符，加上注释，定义的接口变量一定是与接口方法相关的应用级常量

Interface声明的成员方法，默认就是public abstract

Interface声明的变量默认就是public static final

### 枚举名称定义

类名带上Enum后缀

成员名称大小，下划线分隔

正例：枚举名字：DealStatusEnum；成员名称：SUCCESS / UNKOWN\_REASON

### 各层命名规约

* 方法命名

获取单个对象 getXxx

获取多个对象listXxx

获取统计值 countXxx

插入saveXxx / insertXxx

删除 removeXxx / deleteXxx

修改 updateXxx

* 领域模型命名:POJO为统称，禁止命名成xxxPOJO

数据表xxx对象：xxxDO

数据传输对象：xxxDTO

页面xxx展示对象xxxVO

## 常量定义

### 禁止魔法值

反例：String key="Id#taobao\_"+tradeId；

### Long型赋值

必须大写

正例：Long a = 2L

反例：Long a = 2l

## 格式规约

### 保留字括号空格

if/for/while/switch/do等保留字与左右括号之间都必须加空格

### 换行

相对上一行缩进一个tab

运算符，点（.）符号与下文一起换行

括号前，逗号不换行

正例：sb.append("zi").append("xin")…

.append("huang")……

+ aaa + …;

反例：sb.append("zi"). //.号

append("xin"). append

("huang"); //括号前不换行

//参数很多的方法调用也超过120个字符，逗号后才是换行处

method(args1, args2, args3, ...

, argsX);

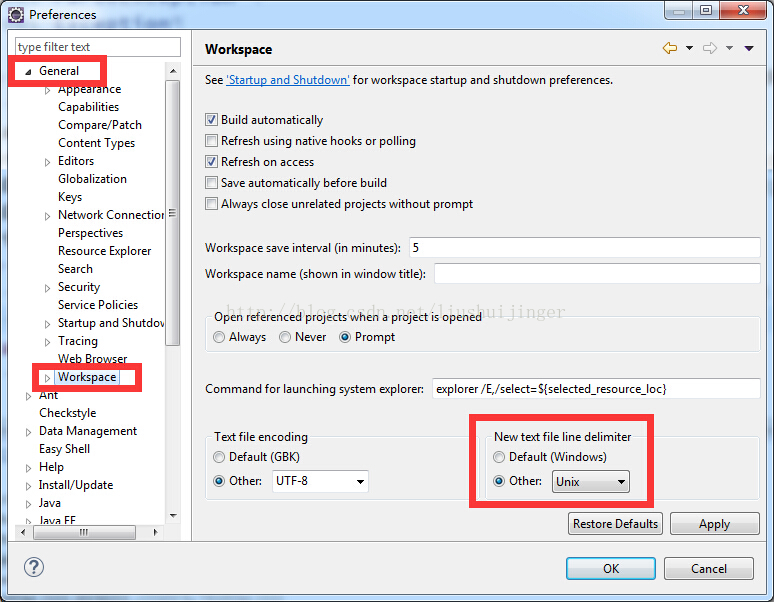
### 方法定义传参

多个参数逗号后边必须加空格

### IDE文件换行符

换行符使用Unix格式，不要使用windows格式

|  |  |
| --- | --- |
| **系统** | **换行符** |
| Windows | \n\r |
| Unix | \n |
| Mac | \r |



## OOP规约

### 访问静态变量/静态方法

直接类名访问

避免通过对象引用访问，无谓增加编译器解析成本

### @Override

覆写方法加@Override注解，编译检查，可以判断是否覆盖成功

### 接口签名修改

原则上不允许修改，接口过时使用@Deprecated注解，并注明新接口是什么

### 值比较

使用工具类java.util.Objects.equals(a, b);

包装类务必使用equals方法

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。

Long short类型也是如此

public static Integer valueOf(int i) {

assert IntegerCache.high >= 127;

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

### 基本数据类型 vs 包装数据类型

所有的POJO类属性必须使用包装数据类型

RPC(远程过程调用协议)方法的返回值和参数必须使用包装数据类型

所有的局部变量推荐使用基本数据类型

### POJO不设属性默认值

反例：某业务的DO的gmtCreate默认值为new Date();但是这个属性在数据提取时并没有置入具体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间。

### 序列化

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

注意serialVersionUID不一致会抛出序列化运行时异常

### 业务逻辑初始化

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

### POJO toString()方法

都加上toString()，如有继承，前面加上super.toString()

抛出异常时可以调用toString方法打印属性值，方便排查问题

## 集合处理

### Map / Set key值重写hashCode和equals

### ArrayList vs subList

ArrayList的subList结果不可强转成ArrayList，

subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList ，而是 ArrayList 的一个视图，对于SubList子列表的所有操作最终会反映到原列表上

public List<E> subList(int fromIndex, int toIndex) {

subListRangeCheck(fromIndex, toIndex, size);

return new SubList(this, 0, fromIndex, toIndex);

}

private class SubList extends AbstractList<E> implements RandomAccess

Arrays.asList() 返回的也是Arrays内部类，实际上后台数据仍为数组

### sublist

在subList场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、删除均产生ConcurrentModificationException 异常

Arraylist中有成员变量

protected transient int modCount = 0;

当在原集合中修改元素个数后（结构性修改structurally modified，增删元素，更新元素就不会），modCount++，记录下修改的次数

而在subList，由于记录下的modCount还是原来的modCount，与原集合的没有同步，就会抛出异常

另，若在sublist中有结构性修改，那就可以反映到原集合中，不会报错，反之就不行

### 集合转数组 toArray(T[] array)

转数组使用toArray(T[] array)

若使用toArray()，返回的Object[] 类型，在做转型时可能出现ClassCastException

List<String> list = new ArrayList<String>(2);

list.add("guan");

list.add("bao");

//指定size，否则默认返回的多余数组值为null

String[] array = new String[list.size()];

array = list.toArray(array);

### 集合通配符

泛型通配符<? extends T>来接收返回的数据，此写法的泛型集合不能使用add方法。 说明：苹果装箱后返回一个<? extends Fruits>对象，此对象就不能往里加任何水果，包括苹果。

增删集合元素

不要在foreach循环里进行元素的remove/add操作，可以用下标值/ Iterator增删

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);

}

}

正例： 单线程环境

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

while(it.hasNext()){

String temp = it.next();

if(删除元素的条件){

it.remove(); //not a.remove();

}

}

多线程：CopyOnWriteArrayList，CopyOnWriteArraySet，每次add,remove等所有的操作都是重新创建一个新的数组，再把引用指向新的数组。这样便可实现线程安全，也不需modCount域，所以对其进行remove和add不会抛出并发异常

Iterator 中内部类实现

public void remove() {

if (lastRet < 0)

throw new IllegalStateException();

checkForComodification();

try {

ArrayList.this.remove(lastRet);

cursor = lastRet;

lastRet = -1;

expectedModCount = modCount;

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

* COW（Copy-On-Write）

CopyOnWrite容器即写时复制的容器。当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。

这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

* fail-fast（快速失败机制）

是Java集合的一种是为了防止多线程修改集合造成并发问题的机制。当多个线程对集合进行结构上的改变的操作时，modCount与expectedModCount不同会产生异常。

### 集合初始化大小

数组类以插入元素数量指定大小，map set类型则需比实际数量稍大一点（maybe 10%）

[what-is-the-optimal-capacity-and-load-factor-for-a-fixed-size-hashmap](http://stackoverflow.com/questions/7115445/what-is-the-optimal-capacity-and-load-factor-for-a-fixed-size-hashmap/7154181#7154181)

### K/V存储null

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **集合类** | **Key** | **Value** | **Super** | **说明** |
| Hashtable | not null | not null | Dictionary | 线程安全 |
| ConcurrentHashMap | not null | not null | AbstractMap | 线程局部安全 |
| TreeMap | not null | null | AbstractMap | 线程不安全 |
| HashMap | null | null | AbstractMap | 线程不安全 |

## 并发处理

### 单例线程安全

获取单例对象要线程安全。在单例对象里面做操作也要保证线程安全。

http://blog.csdn.net/cselmu9/article/details/51366946

### 线程由线程池提供

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

http://www.cnblogs.com/dolphin0520/p/3932921.html

### 日期类SimpleDateFormat & DateUtils

SimpleDateFormat 是线程不安全的类，一般不要定义为static变量，如果定义为static，必须加锁，或者使用自定义的DateUtils工具类。

正例：注意线程安全，使用DateUtils。亦推荐如下处理：

private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>(){

@Override

protected DateFormat initialValue() {

return new SimpleDateFormat("yyyy-MM-dd");

}

https://my.oschina.net/leejun2005/blog/152253

### 锁的使用

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

### 并发修改

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。

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。

乐观锁：每次读取时认为数据不会修改，读写不加锁，更新时查看版本号是否一致

悲观锁：每次读取数据时认为数据可能会被修改，故读写加锁

### 多线程定时任务

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。

### 不使用Executors，用ThreadPoolExecutor创建线程池

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。

Executors各个方法的弊端：

1. newFixedThreadPool和newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至OOM。

1. newCachedThreadPool和newScheduledThreadPool:

主要问题是线程数最大数是Integer.MAX\_VALUE，可能会创建数量非常多的线程，甚至OOM

### 线程/线程池创建时指定有意义的名称

方便出错时回溯

### CountDownLatch 必须执行 countDown方法退出

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方法可以执行，避免主线程无法执行至countDown方法，直到超时才返回结果

注意，子线程抛出异常堆栈，不能在主线程try-catch到。

### 双重检查锁 volatile

[The "Double-Checked Locking is Broken" Declaration](http://stackoverflow.com/questions/3578604/how-to-solve-the-double-checked-locking-is-broken-declaration-in-java)

<http://freish.iteye.com/blog/1008304>

class Foo {

private volatile Helper helper = null;

public Helper getHelper() {

if (helper == null) {

synchronized(this) {

if (helper == null) helper = new Helper();

}

}

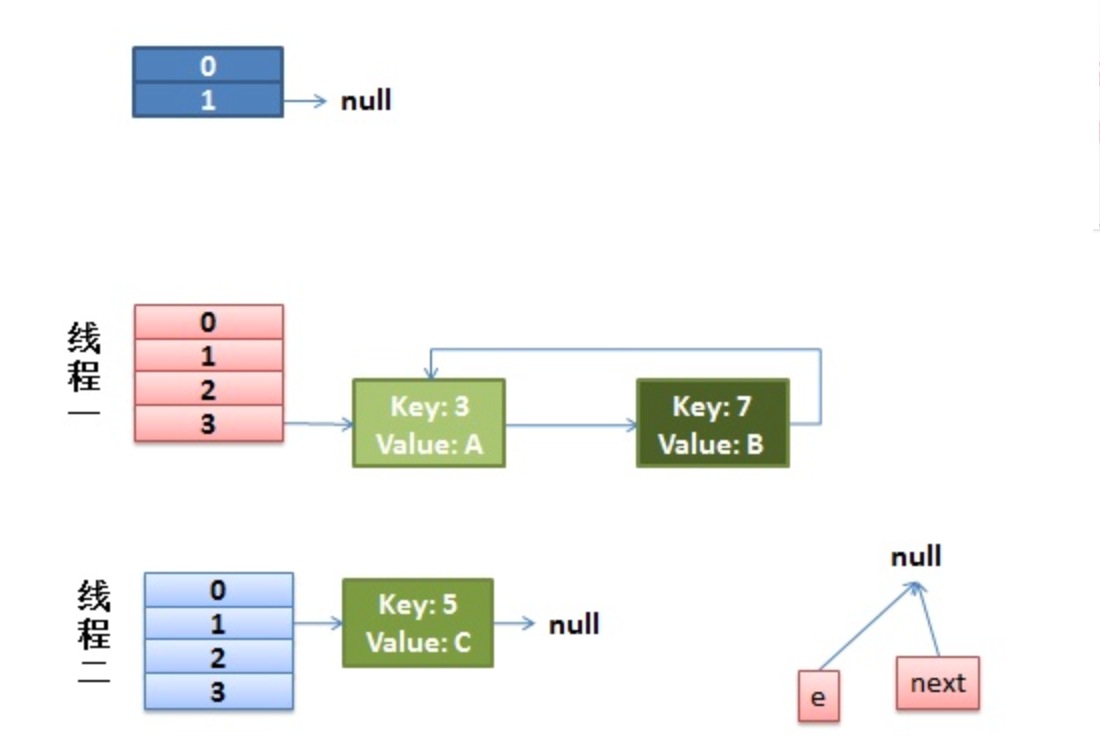
return helper;

}

### 【参考】注意HashMap的扩容死链，导致CPU飙升的问题。

在多线编程环境下，由于没有同步，就可能导致hashmap resize时出现死循环，

<https://my.oschina.net/hosee/blog/673521>



### ThreadLocal无法解决共享对象的更新

【参考】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有的，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。

## 控制语句

### Switch

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

### If else 判断

Condition 条件尽量简单，复杂则抽出

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;

如果使用要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

### 参数校验

* 需要校验：

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。

* 不需要校验

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

### Javadoc注释

类、类属性、类方法、抽象方法

### 单行注释 //

方法内部，被注释语句上方另起一行

### 枚举

每个枚举字段都需注释，说明每个数据项用途

## 其他

### 预编译正则表达式

在使用正则表达式时，利用好其预编译功能

不要在方法体内定义：Pattern pattern = Pattern.compile(规则);

### Bean 属性copy

避免用Apache Beanutils进行属性的copy，Apache BeanUtils性能较差，可以使用其他方案比如Spring BeanUtils, Cglib BeanCopier

### 获取时间

使用System.currentTimeMillis(); 而不是new Date().getTime();

在JDK8中，针对统计时间等场景，推荐使用Instant类

# 异常日志



## 异常处理



### 不捕获RuntimeException子类

不要捕获Java类库中定义的继承自RuntimeException的运行时异常类，如：IndexOutOfBoundsException / NullPointerException，这类异常由程序员预检查来规避，保证程序健壮性。

正例：if(obj != null) {...}

反例：try { obj.method() } catch(NullPointerException e){…}

### Try catch

只Try catch最小非稳定代码，并区分异常类型，分别处理，或者传给上层业务使用者，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

### Finally return

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

当在try块或catch块中遇到return语句时，finally语句块将在方法返回之前被执行。在以下4种特殊情况下，finally块不会被执行：

1）在finally语句块中发生了异常。

2）在前面的代码中用了System.exit()退出程序。

3）程序所在的线程死亡。

4）关闭CPU。

### Null 值返回

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

集合里的元素即使isNotEmpty，取出的数据元素也可能为null

### 抛异常 vs 返回错误码

公司外的http/api开放接口必须使用“错误码”；

应用内部推荐异常抛出

## 日志规约

### 日志保存日期

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

### 异常信息

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，那么往上抛

正例：logger.error(各类参数或者对象toString + "\_" + e.getMessage(), e);

# MYSQL规约



## 建表规约

### 是否字段 0/1

使用is\_xxx的方式命名，数据类型是unsigned tinyint（ 1表示是，0表示否）

任何字段如果为非负数，必须是unsigned

### 小数类型decimal

小数类型为decimal，禁止使用float和double，会损失精度

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

### Text:字段长短超过5000

表中字段储存长度过大，为text类型，独立出来，用主键对应，避免影响其他字段索引效率

## 索引规约

### 唯一索引

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

### Varchar字段索引长度

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

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。

### 索引排序

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。

## SQL规约

### Count

使用count(\*),不使用count(列名)方式，count(\*)会统计值为NULL的行，而count(列名)不会统计此列为NULL值的行

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。

### 不使用存储过程

禁止使用存储过程，存储过程难以调试和扩展，更没有移植性

### TRUNCATE vs DELETE

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。

### Xml配置参数

使用：#{}，#param# 不要使用${} 此种方式容易出现SQL注入

### 动态值判断

<isEqual>中的compareValue是与属性值对比的常量，一般是数字，表示相等时带上此条件；

<isNotEmpty>表示不为空且不为null时执行；

<isNotNull>表示不为null值时执行。