凡乐 JAVA 开发手册

莲花山版　V1.0.0

前 言

《凡乐 Java 开发手册》是凡乐服务端团队经过一年的 Code Review 总结而来的的一些内部编码经验。自去年疫情好转回公司办公以后，我组织了每周一次的跨项目 Code Review。做这件事的初衷与团队的考核无关，而是从公司的实际情况出发——小项目居多，大家各自负责自己手中的代码，太少有机会去阅读他人的代码了。每周一个多小时的代码走读，从最开始我来讲解，到后来每个人轮流主导讲解，代码中一些或漂亮或别扭的地方渐渐得到了大家的共识，录入这本手册中的条目，是我们讨论后一致认可的。

这本手册的制定，也是受到阿里《Java 开发手册》的启发。阿里的手册，很大程度推动了业内 Java 程序员的基本素养，抛开编码规约不谈，一些在编码中的重点知识得到了关注，比如线程池的构造方法、SQL 索引的规约等，条目的背后也是程序员进一步探究学习的引导。阿里的手册以五岳对版本命名，我们打算以莲花山、南山、塘朗山、羊台山、凤凰山、梧桐山等“鹏城五岳”来为手册大版本命名，虽然目前内容远不及阿里的开发手册，“莲花山版”也只是 10 多个简单条目收录，未构成大的结构体系，还算不上“一岳”，但希望未来某一天“梧桐山版”能配得上“鹏城第一峰”的称号。

条目内容并非对阿里手摘抄或是补充，而是出自项目中遇到的真实场景摘录，所以可能出现和阿里手册重复或是不同的理解。暂时抛弃了【强制】【推荐】的条目标注，也未使用【正例】【反例】，每一条目除了描述和说明外，由一组【好味道】+【坏味道】或【新味道】+【老味道】的代码对比组成，也可是单独的【好味道】，其中【坏味道】中产生明确的负作用，应该尽量避免。

代码风格没有绝对的好与坏，码出高效，码出质量，是我们团队协作的共同目标。

悦洋

2021 年 3 月

目 录

一、Java 编码

1. 创建容器类时，若业务场景可知该容器元素个数，推荐传入 initialCapacity 参数。

【好味道】

```
Map<String, Double> gender2AveHeightMap = new HashMap<>(2);
// 存放计算后前三的身高
List<Double> top3HeightList = new ArrayList<>(3);
```

说明：

虽然集合类都有自动扩容机制，但 initialCapacity 参数具明确的语义，合理使用可提高代码可读性。对于已知 size 很大的容器，指定初始化大小也可避免扩容时的性能消耗。

2. 对于已知初始元素的集合，判断集合是否可变，合理使用 Google Guava 库进行初始化。

【好味道】

```
// 可变列表，作为素数缓存，存初始化素数前几项，运算后可调用 add 方法继续加入列表
List<String> primeNumbers = Lists.newArrayList("2", "3", "5", "7", "11", "13");
// 不可变列表，生肖有且只有 12 个，没有 add 方法
List<String> zodiacList = ImmutableList.of("鼠", "牛", "虎", "兔", "龙", "蛇", "
马", "羊", "猴", "鸡", "狗", "猪");
// 不可变 Map，如性别中文和拼音的映射
Map<String, String> gender2pinyin = ImmutableMap.of("男", "nan", "女", "nv");
```

说明：

对于已知部分或全部元素的集合，可使用 Guava 库简化创建，如果集合不可变，推荐使用 Immutable 去创建，避免后续对集合产生错误修改。

【老味道】

```
List<String> genderList = Collections.singletonList("男");
```

说明：

通常当 JSON 响应有特殊约定时，使用该方法可将单个元素包装为列表返回。此列表不可变，没有 set 和 add 方法。“singleton”的命名容易和“单例模式（Singleton Pattern）”混淆，建议使用 Guava 库 ImmutableList 替代。

【坏味道】

```
// Arrays.asList 的错误使用方式，素数列表本意是可变的
List<String> primeNumbers = Arrays.asList("2", "3", "5", "7", "11", "13");
```

说明：

Arrays.asList 的返回对象是一个 Arrays 内部类，并非 java.util.ArrayList，和传入的数组共享内存。可以 set，但不能 add。

3. **字符串、普通对象、集合类对象调用 equals 方法前需要校验自身是否为 null，避免 NPE 异常。**

【坏味道】

```
// NPE 异常
String s = null;
if (s.equals("Hello")) {
}
```

【老味道】

```
String s = null;
// 优先判断字符串对象是否为 null
if (s != null && s.equals("Hello")) {
}
// 也可将常量字符串前置，避免产生 NPE
if ("Hello".equals(s)) {
}
```

【好味道】

```
String s = null;
// 使用 JDK 自带类库，并且不局限于字符串对象间的比较
if (Objects.equals(s, "Hello")) {
}
// 使用三方库 Apache Commons Lang，语义更明确是在对比字符串
if (StringUtils.equals(s, "Hello")) {
}
// 空串校验
String blank = "    ";
// 空格串，返回 false
if (StringUtils.isEmpty(blank)) {
}
// 返回 true
if (StringUtils.isBlank(blank)) {
}
// 使用三方库 Apache Commons Collections，用作集合类的比较
// 返回 true
CollectionUtils.isEqualCollection(Lists.newArrayList(1, 2),
ImmutableList.of(1, 2));
// 注意，返回 false，如没有对数组的操作，尽量避免使用 Arrays.asList
CollectionUtils.isEqualCollection(Arrays.asList(new int[]{1, 2}),
ImmutableList.of(1, 2));
```

说明：

使用类库中的 equals 方法，自带了 null 值判断，不会产生 NPE 且语义更加明确。

4. 所有整型包装类对象之间值的比较，全部使用 equals 方法比较。

【坏味道】

```
Integer a = 100;
Integer b = 100;
Integer c = 1000;
Integer d = 1000;
// 小于 127 会执行，但不要这样写
if (a == b) {
```

```
}
// 不会执行
if (c == d) {
}
```

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 值使用 == 进行判断返回 true，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。

5. **浮点数之间的等值判断，基本数据类型不能用 == 来比较，包装数据类型不能用 equals 来判断。**

【坏味道】

```
double d1 = 1.0;
double d2 = 1.0;
// 返回 false，这样判断并不相等
if (d1 == d2) {

}
double d3 = 1.0f - 0.9f;
double d4 = 0.9f - 0.8f;
// 转换为包装类，业务预期返回 true，实际返回 false
if (Double.valueOf(d3).equals(d4)) {
    // 不会执行判断内逻辑
}
if (Objects.equals(a, b)) {
    // 不会执行判断内逻辑
}
```

【好味道】

```
double a = 1.0;
double b = 1.0;
```

```
// 返回 true，手动指定比较精度
double precision = 0.01;
if (Math.abs(a - b) < precision) {

}
// true
boolean case1 = Math.abs(Math.PI - 3.14) < 0.01;
// false
boolean case2 = Math.abs(Math.PI - 3.14) < 0.001;
```

说明：

浮点数采用“尾数+阶码”的编码方式，类似于科学计数法的“有效数字+指数”的表示方式。

二进制无法精确表示大部分的十进制小数。

6. 对 int 类型的参数进行运算校验，要对 32 位取值范围保持敏感。

【坏味道】

```
int price = 1000_000;
int count = 80_000;
int account = 1000;
// 整数范围溢出 BUG，行为未知，可能返回 false
if (price * count > account) {

}
```

【好味道】

```
int price = 1000_000;
int count = 80_000;
int account = 1000;
// 对其中一个乘数强制转换
if ((long) price * count > account) {

}
```

说明：

int 型整数参与加法乘法运算时，记得潜在的范围溢出问题，尤其在对不受信任的客户端进行传参校验的场景。

7. Comparator 实现类要满足自反性，传递性，对称性这三个条件。

说明：

三个条件如下：

1）自反性：x，y 的比较结果和 y，x 的比较结果相反。

2）传递性：x>y，y>z，则 x>z。

3）对称性：x=y，则 x，z 比较结果和 y，z 比较结果相同。

如不满足，在 JDK7 版本及以上， Arrays.sort、Collections.sort 在数据量大时会抛 IllegalArgumentException 异常。

【坏味道】

```
String s1 = "abcdefg";
String s2 = "xyz";
ArrayList<String> list = Lists.newArrayList(s1, s2);
// 按字符串长度从小到大排序，当字符串长度相等时不满足自反性。几条数据不会报错，但当数据较多时会产生 IllegalArgumentException 异常
list.sort(new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() > s2.length() ? 1 : -1;
    }
});
```

【老味道】

```
// 等值时返回 0，满足自反性
list.sort(new Comparator<String>() {
    @Override
```

```
    public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return 0;
        }
        return s1.length() > s2.length() ? 1 : -1;
    }
});

// 数值型的比较可以直接相减，与上面的写法等价，更优雅
list.sort(new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

【新味道】

```
// 使用 Lambda 表达式代替匿名内部类
list.sort((a, b) -> a.length() - b.length());
// 使用函数式接口，语义更加明确
list.sort(Comparator.comparingInt(String::length));
```

8. 返回值为集合类的方法，不要返回 null，而是用空集合表示“无”这种语义。

【好味道】

```
public List<Integer> primeList(int topN) {
    if (topN <= 0) {
        // 返回空的 ArrayList，该 List 后续可能涉及 add 操作
        return new ArrayList<>();
        // 若明确集合不可变返回 Collections.emptyList()单例对象更好
        // return Collections.emptyList();
    }
    else {
        // ...
    }
}
// ...
List<Integer> primeList = primeList(topN);
// 即使这里没有 isEmpty 判断，for 循环时也不用担心 NPE 异常
if (CollectionUtils.isEmpty(primeList)) {
```

```
    for (int prime : primeList) {

    }
}
```

说明：

当返回值为集合类的方法内部查询为空时，返回空集合而不是 null，这样外部调用方进行循环遍历时不需要额外校验。

此处和阿里手册有出入（阿里手册明确“防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 null 的情况。”）

我们的观点是接口设计者有义务减少调用方潜在出错的可能，这点与调用者的校验责任并不冲突。

另外，最小化可变性是很好的编码思想，Collections.emptyList()在合理的场景下可以使用。有两点好处，1：防止调用者错误使用 add 方法，2：单例，如果方法在循环中被调用，可减少创建对象和垃圾回收的开销。

9. 方法中需要返回多值时，根据实际情况，选用合理的方案。

说明：

Java 语法上没有直接的多值返回，多值返回大致有以下的做法：

1、返回数组或集合对象。

2、返回 Map。

3、封装 POJO 类返回。

4、封装成 JSON 类对象返回。

5、通过的型如 XXXHolder 的“引用”类型参数传递返回。

6、通过类似 Pair/Triple 的“元组”形式返回。

【老味道】

```
public int[] readConfigCase1() {
    int config1 = 1;
    int config2 = 3;
    return new int[]{config1, config2};
}

public Object[] readConfigCase2() {
    int config1 = 1;
    String config2 = "33";
    return new Object[]{config1, config2};
}

public void readConfigCase3(StringHolder config1, IntHolder config2) {
    config1.value = "config";
    config2.value = 1;
}

// ...
// 调用者按约定依次取值
int[] configs = readConfigCase1();
int config1_1 = configs[0];
int config1_2 = configs[1];

// 调用者需要强制类型转换，硬编码，不推荐
Object[] configs = readConfigCase2();
String config2_1 = (String)configs[0];
int config2_2 = (Integer)configs[1];

// 调用者传入 XXXHolder 的“引用”类型参数
StringHolder stringHolder = new StringHolder();
IntHolder intHolder = new IntHolder();
readConfigCase3(stringHolder, intHolder);
String config3_1 = stringHolder.value;
int config3_2 = intHolder.value;
```

【新味道】

```
// Apache Commons 的 Pair 工具类
public Pair<String, Integer> readConfig4() {
    return Pair.of("Tom", 18);
}
// 调用者无需强制类型转换
Pair<String, Integer> pair = readConfig4();
String config4_1 = pair.getLeft();
int config4_2 = pair.getRight();

// Lambda 表达式中使用多值返回
List<Row> rowList = new ArrayList<>(3);
rowList.add(new Row(1, 1));
rowList.add(new Row(1, 1));
rowList.add(new Row(1, 3));
// fieldA、fieldB 组合出现的次数
Map<Pair<Integer, Integer>, Long> comboCountMap = rowList.stream()
       .collect(groupingBy(row -> Pair.of(row.getFieldA(), row.getFieldB()),
counting()));
```

说明：

编码时选择哪种多值返回方案，有以下考量：

1、返回类型是否相同，如果不同不适用数组形式返回。

2、Map 和 POJO 按实际情况灵活选择，Map 适用于 HttpClient 调用这种泛化性高的 API，而 POJO 更多是在 RPC 调用中封装。

3、如果方法会出现在 Lambda 表达式的链式调用之中，推荐使用 Apache Commons 的 Pair/Triple 工具类。

4、如果经常有返回值超过三个的情况，可以自己封装 Turple 工具类（注意拼写不是 Triple，Triple 是“三倍”，而 Turple 是“元组”的意思）。

10. 使用 Lambda 操作时，推荐引入静态引入 java.util.stream.Collectors 包。

【好味道】

```
// 静态引入 Collectors 包
import static java.util.stream.Collectors.*;
// ...
ArrayList<Integer> list = Lists.newArrayList(0, 1, -1, -2, 3, 4);
// 引入前
list.stream().filter(a -> a < 0).collect(Collectors.toList());
// 引入后
list.stream().filter(a -> a < 0).collect(toList());
```

说明：

Collectors 下有很多常用的静态方法，在 Lambda 表达式的链式调用中去掉冗余的类前缀，可使代码更加美观易读。

11. 构造函数内不要写任何业务逻辑，尤其注意不要在构造函数内启动线程。

【坏味道】

```
// 错误启动线程
public class ThisEscape {
    public ThisEscape() {
        new Thread(new EscapeRunnable()).start();
        // ...
    }

    private class EscapeRunnable implements Runnable {
        @Override
        public void run() {
            // 通过 ThisEscape.this 就可以引用外围类对象，但是此时外围类对象可能还没有构造完成，即发生了外围类的 this 引用逃逸
        }
    }
}
```

```
// 错误的注册监听器方式
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
                new EventListener() {
                    public void onEvent(Event e) {
                        // 这里就可以使用未完全构造的 this 对象了
                        doSomething(e);
                    }
                }
        );
    }
}
```

【好味道】

```
public class SafeThread {
    private Thread t;
    public SafeThread() {
        t = new Thread(new EscapeRunnable());
        // ...
    }

    public void init() {
        t.start();
    }

    private class EscapeRunnable implements Runnable {
        @Override
        public void run() {
            // SafeThread.this 就可以引用外围类对象,此时可以保证外围类对象已经构造完成
        }
    }
}

public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
```

```
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

说明：

除了对成员变量的直接赋值外，构造函数内不要有任何业务逻辑，包括打印日志，初始化运算等。对象创建过程存在指令重排，“This 逃逸”等问题，不是线程安全的。

如果想在构造函数中注册一个事件监听器或启动线程，可以使用一个私有的构造函数和一个公共的工厂方法，从而避免不正确的构造过程

12. 注意 Optional 的 orElse 和 orElseGet 方法区别，不要在 orElse 里 new 一个对象，而是使用单例的默认对象。

【坏味道】

```
// 业务逻辑是从列表获取一个 id 大于 10 的角色，列表里没有返回默认角色，无论列表中是否存在大于 10 的角色，都会调用构造函数，输出“create a role”
Role role = members.stream()
        .filter(r -> r.getId() > 10)
        .findAny()
        .orElse(new Role());
// ...角色类
public class Role {
    private Integer id;
    private String name;
    private static final Role defaultRole = new Role(0, "default");
    //...

    public Role() {
        // ...示例说明
```

```
        System.out.println("create a role");
    }

    public Role(Integer id, String name) {
        this.id = id;
        this.name = name;
    }

    public static Role getDefault() {
        return defaultRole;
    }
}
```

【好味道】

```
List<Integer> numbers = Lists.newArrayList(1, 2, 3, -1, -2, -3);
Integer number = null;
// 基础数据类型，则两种方式区别不大
number = numbers.stream().findAny().orElse(0);
number = numbers.stream().findAny().orElseGet(() -> 0);

// 若找到 ID 大于 10 的角色 orElseGet 不会创建
role = members.stream()
        .filter(r -> r.getId() > 10)
        .findAny()
        .orElseGet(() -> new Role());
// 可使用默认单例
role = members.stream()
        .filter(r -> r.getId() > 10)
        .findAny()
        .orElse(Role.getDefault());

// 类似 Map 的 getOrDefault 含义
Map map = new HashMap();
map.getOrDefault(key, defaultValue);
Optional.ofNullable(map.get(key)).orElse(defaultValue);
Optional.ofNullable(map.get(key)).orElseGet(() -> defaultValue);
```

说明：

orElseGet 传递的参数是函数式接口实例，orElse 传递的参数是对象实例，在 orElse 中 new 对象和条件无关，当条件满足时，对象创建后直接丢失引用，造成不必要的开销。

13. 使用随机数时，不要随处 new Random()，若不需要手动设置随机数种子，使用 ThreadLocalRandom 更好。

【坏味道】

```
// 这种方式创建线程池不好，这里只是做示例
ExecutorService pool = Executors.newFixedThreadPool(1000);
pool.execute(() -> {
    // ...不必要的 Random 对象创建
    Random random = new Random();
    random.nextInt(100);
});
```

【老味道】

```
// 用 Math.random 生成 [0, 100) 的随机整数并不优雅
int randomInt = (int)(100 * Math.random());
// 固定随机种子，以及线程安全的例子
Random random1 = new Random();
Random random2 = new Random();
// 某些业务场景需要指定固定随机种子（如生成 100 万随机兑换码，可以只记录种子）
random1.setSeed(1L);
random2.setSeed(1L);
List list1 = Lists.newArrayList();
List list2 = Lists.newArrayList();
List list3 = Lists.newCopyOnWriteArrayList();
for (int i = 0; i < 100; i++) {
    list1.add(random1.nextInt(100));
    list2.add(random2.nextInt(100));
}
Random random3 = new Random();
random3.setSeed(1L);
List list = Lists.newArrayList();
Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 50; i++) {
        list3.add(random3.nextInt(100));
```

```
            Thread.yield();
        }
    });
    Thread thread2 = new Thread(() -> {
        for (int i = 0; i < 50; i++) {
            list3.add(random3.nextInt(100));
            Thread.yield();
        }
    });
    thread1.start();
    thread2.start();
    Thread.sleep(100);
    // true，因为随机种子相同
    CollectionUtils.isEqualCollection(list1, list2);
    // true，Random 线程安全
    CollectionUtils.isEqualCollection(list1, list3);
```

【新味道】

```
// 随机生成 [0,100) 间服从均匀分布的整数，调用方便，线程安全
ThreadLocalRandom.current().nextInt(100);
```

说明：

Random 类本身线程安全，无需随处创建，但每次随机有 CAS 判定，性能相比 ThreadLocalRandom 略差，且要自己封装单例。ThreadLocalRandom 不能手动设置 seed。Math.random() 使用的是 Random 单例，返回的是 [0，1）的 double，也不能手动设置 seed。

14. 对日期时间工具类的封装，优先考虑 JDK8 的 java.time 包而不是 java.util.Date。

【坏味道】

```
// 这种线程池创建方式不好，这里只是用作示例
ExecutorService pool = Executors.newFixedThreadPool(10);
for (int i = 0; i < 1000; i++) {
    // SimpleDateFormat 不是线程安全的，此处会出现 java.lang.NumberFormatException
```

```
异常
    pool.execute(() -> DateUtil.parse(DateUtil.now()));
}

// ...
public static class DateUtil {
    // 不是线程安全的，错误的用法
    public static final SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

    public static String now() {
        return df.format(new Date());
    }

    public static Date parse(String time) {
        try {
            return df.parse(time);
        } catch (ParseException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

【老味道】

```
public static class SafeDateUtil {
    // 不要每次使用都创建，线程私有即可，不会出现并发问题
    public static final ThreadLocal<SimpleDateFormat> df =
ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));

    public static String now() {
        return df.get().format(new Date());
    }

    public static Date parse(String time) {
        try {
            return df.get().parse(time);
        } catch (ParseException e) {
            e.printStackTrace();
            return null;
        }
```

```
    }
}
```

【好味道】

```
// 使用 JDK8 的 java.time 包封装时间日期工具类，线程安全且代码更加优雅
public static class DateUtil {
    public static final DateTimeFormatter format =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    public static String now() {
        return LocalDateTime.now().format(format);
    }

    public static LocalDateTime parse(String time) {
        return LocalDateTime.from(format.parse(time));
    }
    // ...
}
```

说明：

SimpleDateFormat 创建开销大且不是线程安全的，其 API 也包含了很多臃肿的异常设计。

新项目代码尽量不再使用 java.util.Date，积极使用 JDK8 提供的更健壮时间/日期包 java.time。

15. 无特殊原因，不要修改测试环境 Linux 服务器系统时间。功能测试时，由开发提供接口修改服务进程内的时间。

【好味道】

```
public static class DateUtil {
    public static final DateTimeFormatter format =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    private volatile static long offsetSecond;

    // 外部接口调用设置测试时间
    public static void setOffsetSecond(long offsetSecond) {
```

```
        DateUtil.offsetSecond = offsetSecond;
    }

    // 恢复时间
    public static void resetOffsetSecond() {
        setOffsetSecond(0);
    }

    // 获取时间的方法
    public static String now() {
        return LocalDateTime.now().plusSeconds(offsetSecond).format(format);
    }

    public static LocalDateTime parse(String time) {
        return LocalDateTime.from(format.parse(time));
    }
    // ...
}
```

说明：

测试环境不止一个项目在运行，应避免修改系统时间。需要修改时间的测试用例，由开发提供粒度最小的修改方案。

16. 在使用阻塞等待获取锁的方式中，必须在 try 代码块之外，并且在加锁方法与 try 代码块之间没有任何可能抛出异常的方法调用，避免加锁成功后，在 finally 中无法解锁。

【坏味道】

```
Lock lock = new ReentrantLock();
try {
    // 如果此处抛出异常，则直接执行 finally 代码块
    doSomething();
    // 无论加锁是否成功，finally 代码块都会执行
    lock.lock();
    doOthers();
} finally {
```

```
    lock.unlock();
}
```

【好味道】

```
Lock lock = new ReentrantLock();
// ...
lock.lock();
try {
    doSomething();
    doOthers();
} finally {
    lock.unlock();
}
```

说明：

此条目摘抄自阿里的手册，我们项目中发现有错误的用法。

有三种异常情况：

1、如果在 lock 方法与 try 代码块之间的方法调用抛出异常，那么无法解锁，造成其它线程无法成功获取锁。

2、如果 lock 方法在 try 代码块之内，可能由于其它方法抛出异常，导致在 finally 代码块中，unlock 对未加锁的对象解锁，它会调用 AQS 的 tryRelease 方法（取决于具体实现类），抛出 IllegalMonitorStateException 异常。

3、在 Lock 对象的 lock 方法实现中可能抛出 unchecked 异常，产生的后果与说明二相同。

二、MySQL 数据库

1. 表创建推荐直接使用 DDL 语句，DDL 语句具有可读性，建议使用工具格式化对齐(可使用 Druid 的 SQL 格式化工具)。

【好味道】

```
CREATE TABLE `demo_table` (
    `id` bigint UNSIGNED NOT NULL AUTO_INCREMENT,
    `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间
',
    `edit_time` datetime NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT '更新时间',
    CONSTRAINT `pk_id` PRIMARY KEY (`id`)
) ENGINE = InnoDB AUTO_INCREMENT = 1 CHARSET = utf8mb4 COMMENT '描述';
```

说明：

JPA 等框架拥有自动建表功能，但用注解代码生成和手写 DDL 一样可读性强的建表语句并不方便。手写的 DDL 语句具有更清晰的说明功能。

上面的通用默认数据库建表语句有以下几点考量：

1、明确使用 InnoDB 引擎（无特殊需求时默认）。

2、表必备三字段：id、create_time、edit_time，其中自增主键 id 无业务含义，但对性能有重要意义。

3、使用 COMMENT 对字段进行备注，有文档意义。

4、无特殊情况字符集使用 utf8mb4，避免 Emoji 字符无法存储的情况。

2. 建表、初始化语句存放于项目约定目录下，并在 README.md 文件中说明。

说明：

README.md 文件需要对项目启动部署进行说明，其中也包括数据库的初始化。

三、项目结构

1. **在项目根目录下要有 README.md 文件对项目进行简要说明，其中要包括项目启动的具体操作流程。**

说明：

README.md 文件是他人观看项目时第一时间要阅读的内容，需要用心编写，清晰完善的流程描述能使团队协作更加高效。

附 1：版本历史

版本名	版本号	发布时间	轮值编写人	备注
莲花山版	V1.0.0	2021-03	悦洋	初始 Demo 版本

附 2：说明

条目中的 Java 编码，如无特殊说明，均基于 JDK8 并默认引入第三方常用类库 Google Guava，Apache Commons。依赖如下：

```
<properties>
    <guava>30.0-jre</guava>
    <commons-lang3>3.11</commons-lang3>
    <commons-collections4>4.4</commons-collections4>
</properties>
<dependencies>
<dependency>
    <groupId>org.apache.commons</groupId>
```

```
    <artifactId>commons-lang3</artifactId>
    <version>${commons-lang3}</version>
</dependency>

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-collections4</artifactId>
    <version>${commons-collections4}</version>
</dependency>

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>${guava}</version>
</dependency>
</dependencies>
```