Java best practice

凯威老师

大纲

- 减少boilerplate
- Reuse
- 最小化可变性
- 声明式编程
- DI解耦
- 可测试的代码（单元测试，集成测试）
- Code review 标准
- Naming
- Comment
- Build on exception

大纲

- 改善已有的代码

减少boilerplate

Boilerplate Codes:

- sections of code that have to be included in many places with little or no alteration
- the programmer must write a lot of code to do minimal jobs
- In object-oriented programs, classes are often provided with methods for **getting and setting** instance variables. The definitions of these methods can frequently be regarded as boilerplate

减少boilerplate - Lombok

Vanilla Java:

```
public class Sample {
      private final String x;
      private final int y;

     public Sample (final String x, final int y) {
            this.x = x;
            this.y = y;
     }

     public String getX() {
            return this.x;
      }

      public String getY() {
            return this.y;
      }

}
```

With Lombok:

```
@RequiredArgsConstructor
@Getter
public class Sample {
      private final String x;
      private final int y;
}
```

减少boilerplate

- Guava
  - Reduces boilerplate when generating standard hashable and comparable Java object.
- Common.lang
  - 各种util
- Intellij
  - lambda expressions recommendation
  - Automatically add “this”, “final” keywords

Reuse

- 使用injection
  - Dependency, Metrics, Logging, Lombok
- 使用common libs
  - Google Guava, Apache Commons, Retry Libs
- Avoid duplicated code

Reuse

```
while(true) {
        try {
                client.execute(req);
                break;//成功跳出
        } catch (Exception ex) {
                if (retry == 0) {...}
                if (retry++ ==maxRetry)
{...}
        }
}
```

- 有时自己造的轮子太弱

```
Callable<Boolean> callable = new Callable<Boolean>() {
    public Boolean call() throws Exception {
        return true; // do something useful here
    }
};

Retryer<Boolean> retryer = RetryerBuilder.<Boolean>newBuilder()
        .retryIfResult(Predicates.<Boolean>isNull())
        .retryIfExceptionOfType(IOException.class)
        .retryIfRuntimeException()
        .withStopStrategy(StopStrategies.stopAfterAttempt(3))
        .build();
try {
    retryer.call(callable);
} catch (RetryException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

最小化可变性

- 尽可能使用immutable classes
- **Advantage**
  - 容易设计和实现.
  - 不易出错，简单，安全.
  - Thread safe，不用考虑同步问题
  - 更容易和其他代码集成
- **Disadvantage**
  - Immutable classes 不同的值就要创建不同的实例，过度消耗资源.

最小化可变性

```
public class Sample {
    private String x;

    public Sample () {}

    public Sample (String x)
{
        this.x = x;
    }

    public void setX(String
x) {
        this.x = x;
    }
    public String getX() {
        return this.x;
    }
}
```

VS

```
public class Sample {
    private final String x;

    public Sample (final String x) {
        this.x = x;
    }

    public String getX() {
        return this.x;
    }
}
```

最小化可变性

- 不提供任何修改接口.
- 尽可能使用final private.
- 无setter，只有构造函数
- Use Guava Immutable Collections

最小化可变性

- Use Guava Immutable Collections

```
public static final ImmutableSet<String> COLOR_NAMES = ImmutableSet.of(
  "red",
  "orange",
  "yellow",
  "green",
  "blue",
  "purple");

class Foo {
  final ImmutableSet<Bar> bars;
  Foo(Set<Bar> bars) {
    this.bars = ImmutableSet.copyOf(bars); // defensive copy!
  }
}
```

声明式编程

- 命令式
  - 在表达如何去做，how
- 声明式
  - 在表达我在做什么，what

声明式编程

```
//Find the total of sqrt of first K primes starting with
n
public static double compute(int n, int k) {
        int index = n;
        int count = 0;
        double total = 0;
        while(count < k) {
              if(isPrime(index)) {

                    total += Math.sqrt(index);

                    count++;
              }
              index++;
         }
         return total;
}
```

声明式编程

```
//前K个质数的平方和
// 从n开始
public static double compute(int n, int k) {
	int index = n;
	int count = 0;
	double total = 0;
	while(count < k) {
		if(isPrime(index)) {

			total += Math.sqrt(index);

			count++;
		}
			index++;
	 }
	 return total;
}
```

```
//前K个质数的平方和
// 从n开始
public static double compute (int n, int k) {
	return Stream.iterate(n, e-> e + 1)
				.filter(test::isPrime)
				.mapToDouble(Math::sqrt)
				.limit(k)
				.sum();
}
```

声明式编程

- 可读性更高
- 减少可变性
- 无状态

DI解耦

- 工厂模式升级实现
- 解耦
- 依赖关系管理
- 易于测试, 容易Mock
- 易于重构

DI解耦/不用DI的例子

```
// An example without dependency injection
public class Client {
    // Internal reference to the service used by this client
    private Service service;

    // Constructor
    public Client() {
        // Specify a specific implementation in the constructor instead of using dependency
        injection
        service = new ServiceExample();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

DI解耦 / DI 例子

```
// An example without dependency injection
public class Client {
        // Internal reference to the service used by this
        client
        private Service service;

        // Constructor
        public Client() {
                // Specify a specific implementation in the
                constructor instead of using dependency injection
                service = new ServiceExample();
        }

        // Method within this client that uses the services
        public String greet() {
                return "Hello " + service.getName();
        }
}
```

```
// An example dependency injection
public class Client {
        @Inject
        private Service service;


        // Method within this client that uses
        the services
        public String greet() {
                return "Hello " + service.getName();
        }
}
```

DI解耦 / Frameworks

- Guice
  - Google开源，轻量级
- Spring
  - Spring框架，广泛使用，资料多

编写可测试的代码

你写的代码真的可测试吗

编写可测试的代码

- 有意义的Constructor
- 容易Mock
- 类，函数划分清晰

编写可测试的代码

```
public void foo() {
        A aService = new A();
        InstA inst = a.readDb();
        B bService = new B();
        InstB ret = b.dispatch(inst);
}
```

- a.readDb() 调用链很长怎么办
- 没有可连的Db怎么办
- 线上编译网络是隔离的怎么办

编写可测试的代码

```
public class FooClass{
        @Inject;
        private A serviceA;
        @Inject;
        private B serviceB;
}
```

```
@InjectMocks
private FooClass fooService

@Mock
private A mockA;

@Mock
private B mockB;

Mockito.doReturn(inst).when(mockA).readDb()
```

- 看，DI很有用吧
- PowerMock for 静态类，静态函数

编写可测试的代码

- 有意义的构造函数
  - 先mock在通过构造函数传递进去
- Setter
  - 用set传递mock对象

编写可测试的代码

- 单元测试的意义：
  - 保证基本逻辑
  - 不惧怕代码频繁修改
  - 大重构
- 集成测试的意义：
  - 减轻QA工作
  - 镜像环境模拟真实的流程
  - 能跨服务测试

Code review标准

- 代码清晰
- 代码正确
- 设计合理
- 可重用
- 可配置性
- 兼容性
- 依赖性
- 错误处理
- 性能

Naming

- Meaningful
- Consistent
- Reflect the domain
- Reflect the operation

Naming

- 类名和对象应该是名词，或者名词短语，不应该是动词
- 方法名应该是动词，或者动词短语

Bad:

- tryToShipOrder()
- OrderStatus: SHIPPED, CANCELLED

Good:

- setOrderCancellationStatus
- OrderCancellationStatus: UNABLE_TO_CANCEL, COMMITED_TO_CANCEL

Comment

```
/**
 * A transition which has been accepted by the head node of the alf bus. If this
 * transition is lucky enough to have made it to a commit node, then it has also
 * been committed by the alf bus.
 *
 * Transitions are immutable. Once a transition has been accepted by an acceptor
 * (head of the chain), and assigned a sequence number, then it is globally
 * unique, with a unique (acceptorId, sn) key. In the event of chain
 * reconfiguration, it can happen that there are several transitions in a chain
 * with the same sequence number, but accepted by different head nodes. The
 * chain algorithm guarantees that only one of those transitions will be
 * committed. Therefore, if you are restricted to the set of committed
 * transitions, the sequence number by itself is a primary key.
 * The outer transition serialization is the same for all transition versions.
 */
public class Transition implements AutoCloseable {
// ...
}
```

- 用文档说明这段代码的目标是解决什么问题，有什么限制。
- 但不鼓励代码里写注释。

Build on exception

- Java处理错误请用异常
- 不要return 各种码
- Java包含两种异常：checked异常和unchecked异常
- 不传递null值，不返回null值
- 如果一定要null， @Nullable

Build on exception

```
try{
        fileInputStream = new
        FileInputStream(file);
        fileInputStream.read();
} catch (IOException e){
        return null;
}
```

```
try{
        fileInputStream = new
        FileInputStream(file);
        fileInputStream.read();
} catch (IOException e){
        throw e;
}
```

- 不要吃掉异常

Build on exception

```
try {
        //some statements
} catch(Exception e){
        //handle here
}
```

```
try {
        //some statements
} catch(FileNotFoundException e){
        //handle here
} catch(IOException e) {
        //..
}
```

记住异常对性能有一定影响

改善已有的代码

- 重新组织你的函数
- 取消嵌套条件判断
- 内联函数
- 内联变量
- 减少重复计算
- 减少magic code
- 替换算法
- builder
- 对性能有损耗的代码

改善已有的代码 / 重新组织你的函数

- 提炼函数

```
String name =
request.getParameter("Name");
if( name != null && name.length() > 0 ){
        ......
}

String age =
request.getParameter("Age");
if( age != null && age.length() > 0 ){
        ......
}
```

- => StringUtils

```
String name =
request.getParameter("Name");
if( !isNullOrEmpty( name ) ){
        ......
}
String age = request.getParameter("Age");
if( !isNullOrEmpty( age ) ){
        ......
}


private boolean isNullOrEmpty( final String
string ){
        if( string != null && string.length() > 0 ){
                return true;
        }else{
                return false;
        }
}
```

改善已有的代码 / 重新组织你的函数

- 功能细分

```
void printOwing() {
        //print banner
        System.out.println(“*********”);
        System.out.println(“Banner”);
        System.out.println(“*********”);
        //print details
        System.out.println ("name: " + _name);
        System.out.println ("amount " +
        getOutstanding());
}
```

```
void printOwing(){
        printBanner();
        printDetails(getOutstanding());
}

void printBanner(){
        System.out.println( “*********” );
        System.out.println( “Banner” );
        System.out.println( “*********” );
}

void printDetails (double outstanding){
        System.out.println ("name: " + _name);
        System.out.println ("amount " + outstanding);
}
```

- 200行以上大函数
- 删除无用注释

改善已有的代码 / 重新组织你的函数

- 函数的第一规则是要短小，第二规则是还要更短小。
- 函数的缩进层数不该多于一层或两层。
- 函数应该做一件事，做好这件事，只做这一件事。
- DRY(don’t repeat yourself)

  - 第一次先写了一段代码。
  - 第二次在另一个地方写了一段相同的代码，你已经有消除和提取重复代码的冲动了。
  - 再次在另一个地方写了同样的代码，你已忍无可忍，现在可以考虑提取和消除重复代码了。

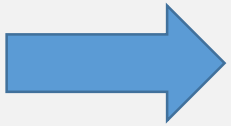
改善已有的代码 / 取消嵌套条件判断

```
double getPayAmount() {
        double result;
        if (_isDead) result = deadAmount();
        else {
                if (_isSeparated) result = separatedAmount();
                else {
                        if (_isRetired) result = retiredAmount();
                        else result = normalPayAmount();
                };
        }
        return result;
};
```

```
double getPayAmount() {
        if (_isDead) return deadAmount();
        if (_isSeparated) return separatedAmount();
        if (_isRetired) return retiredAmount();
        return normalPayAmount();
};
```

改善已有的代码 / 取消嵌套条件判断

```
public double getPayAmount() {
        double result;
        if (isA) {
                if (isB) {
                        if (isC) {
                                return normalPay();
                        }
                }
        }
        return 0;
```



```
public double getPayAmount() {
        if (! isA) {
                return 0;
        }
        if (! isB) {
                return 0;
        }
        if (! isC)
        ....
};
```

- 内层条件外置，减少嵌套
- 合并判断条件，减少嵌套

```
public double getPayAmount() {
        if (isA&& isB&& isC) {
                return 0;
        }
        ....
};
```

改善已有的代码 / 取消嵌套条件判断

```
if (isA) {
...
} else if (isB) {
...
} else if (isC) {
...
} else if (isD) {
...
}
```

```
switch (condition) {
        case isA:
                ...
                break;
        case isB:
                ...
                break;
        ....
}
```

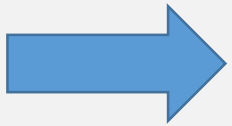
Tableswitch是一张key-value表

O(1)

改善已有的代码 / 内联函数

```
public int getRate() {
        return moreThanFive() ? 2 : 1;
}

public bool moreThanFive() {
        return num > 5;
}
```

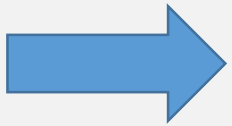


```
public int getRate() {
        return num > 5 ? 2 : 1;
}
```

- 简介
- 信息集中
- 性能

改善已有的代码 / 内联变量

```
double val = pricing.getQuotation();
return val > 1000;
```



```
return pricing.getQuotation() > 1000
```

- 简介
- 信息集中
- 性能

改善已有的代码 / 减少重复计算

```
public double basePrice() {
        return qty * unitPrice;
}
...
if (basePrice() > 1000) {
        return basePrice() * 0.8
}
return basePrice() * 0.9
```

```
double basePrice = qty * unitPrice;
if (basePrice > 1000) {
        return basePrice * 0.8
}
return basePrice * 0.9
```

- 性能问题

改善已有的代码 / 减少magic code

```
if (userAgent.indexOf(“MAC”) {
        …
} else if (userAgent.indexOf(“IE”) {
        …
}
```

```
public static final String MAC = “MAC”;
public static final double
SMOOTH_FACTOR = 0.5;
```

- 代码管理问题
- 多处用，一处维护

```
double tfIdf = termFreq * 1 / (docFreq +
0.5)
```

改善已有的代码 / 替换算法

```
public String findCars(String[] cars) {
        for (String carName : cars) {
                if (carName.equals( “BENZ” )) {
                        return carName;
                }
                else if (carName.equals( “Audi” ) {
                        return carName;
                }
                else if ...
                else if ...
        }
}
```

```
public String findCars(String[] cars) {
        List<String> prefix = Arrays.asList(new
String[] { “BENZ”, “AUDI”, “...” });
        for (...) {
                if (prefix.contains(carName) {
                        ...
                }
        }
}
```

- 简介易读

改善已有的代码 / 替换算法

```
if (case1) {
        strategy1();
} else if (case2) {
        strategy2();
} else if (case3) {
        strategy3();
}
```

public interface Strategy ...

Strategy1, 2, 3实现接口 doWork()..

Map<Integer, Strategy> dispatcher;

用Map来迅速判断

- 代码侵入少
- 性能好

改善已有的代码 / Builder

```
A a = new A();
a.set1(..);
a.set2(..);
...
a.setN(..);
```

```
A a = A.builder()
        .with1(..)
        .with2(..)
        ...
        .withN(..)
        .build();
```

避免的样板式代码

```
b.val1 = a.val1;
b.val2 = a.val2;
```

请用modelmapper

改善已有的代码 / 性能损耗的代码

- 频繁地new（比如在for内）
- 异常处理太多
- 同步方法=>同步代码块，减少同步的代码段
- 少用反射
- 尽量用线程池和连接池
- 少打log
- 少操作大String