Turing College 图灵课堂 | 致敬大师 致敬未来的你

项目实战总结与项目管理经验分享

Monkey

开源框架Flasher作者
京东资深架构师
国美技术委员会成员

课程内容

1、电商项目前面课程总结
2、搜索提示方案详解
3、并发异步加载技术实战
4、线程池管理详解
5、阿里开发手册说明
6、项目管理经验分享

腾讯课堂-图灵课堂

07月14日 晚上20:00

电商的总结：

https://www.processon.com/view/link/60ee7eeaf346fb06e6a7d218

遗漏补缺：

购物车：

离线：

离线购物车针对的是未登录用户。把”**机器设备码**“当成当前用户的身份。

在线：

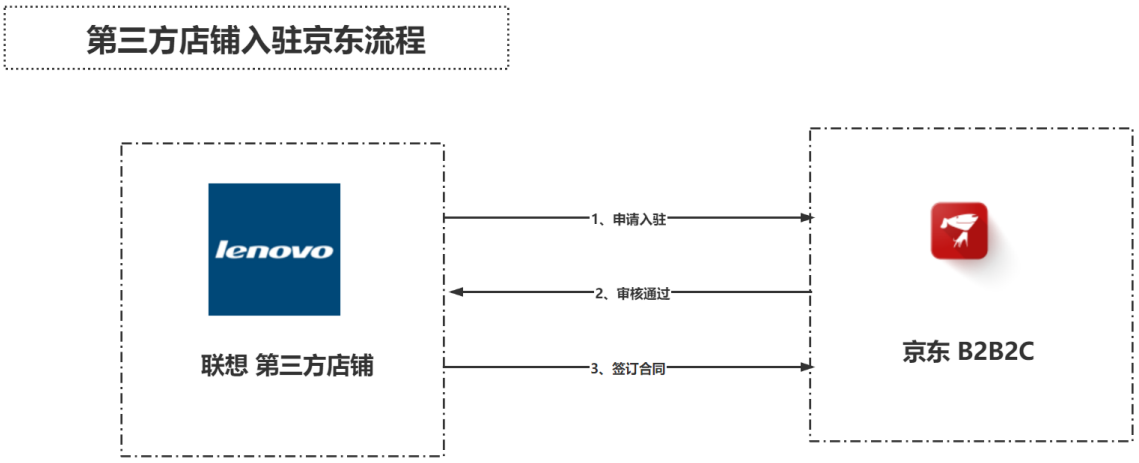
用户登录之后加入的商品购物车。

技术难点：游客账户登录之后，牵扯到离线购物车商品信息的合并。

搜索提示：

有一种功能叫做搜索提示，这个功能如果大家做的话，大家第一想到的是什么？

如果数据量没有几千万，就十几万数据。比如商家入驻这个功能。



D:\workspace

CompletableFuture:

美团外卖、滴滴打车很多场景都需要用到。
当我们在软件下单的时候，付款之后，系统马上就会提示我们下单成功，而这个背后我们来分析下，
有很多地方有数据处理的工作。保存订单到数据库、通知商家接单、通知骑手取货。
整个流程的体验是非常快的，这其中肯定也是异步化的。
我们后端的开发也是一样的，我们不能因为某一次的延迟而让后面所以的请求进行等待。

异步技术：Java 5 Future接口。
使用Future获得异步执行结果时，要么调用阻塞方法get()，要么轮询看isDone()是否为true，这两种方法都不是很好，因为主线程也会被迫等待。

```
1
2  public static void main(String[] args) throws ExecutionException, Interrupted
Exception {
3   ExecutorService executorService = Executors.newFixedThreadPool(2);
4   Future<Integer> result = executorService.submit(() -> {
5   TimeUnit.SECONDS.sleep(5);
6   return 300;
7   });
8   System.out.println(result.get());
9  }
10
11
12  public static void main(String[] args) throws ExecutionException, Interrupted
Exception {
13   ExecutorService executorService = Executors.newFixedThreadPool(2);
14   Future<Integer> result = executorService.submit(() -> {
15   return 300;
16   });
17   while(!result.isDone()){
18   TimeUnit.MILLISECONDS.sleep(100);
19   }
20   System.out.println(result.get());
21  }
```

通过主线程输出语句，我们看到必须要等待线程池执行完成才回执行main线程。
这样的话非常浪费cpu资源的，为什么不能用类似观察者或者监听的模式，当任务完成之后我就能收到结果。这就要涉及到回调函数。

1、同步阻塞调用

即串行调用，响应时间为所有服务的响应时间总和;

2、半异步(异步Future)

线程池，异步Future

使用场景:并发请求多服务。总耗时为最长响应时间;提升总响应时间，但是阻塞主请求线程，高并发时依然会造成线程数过多，CPU上下文切换;

3、全异步(Callback)

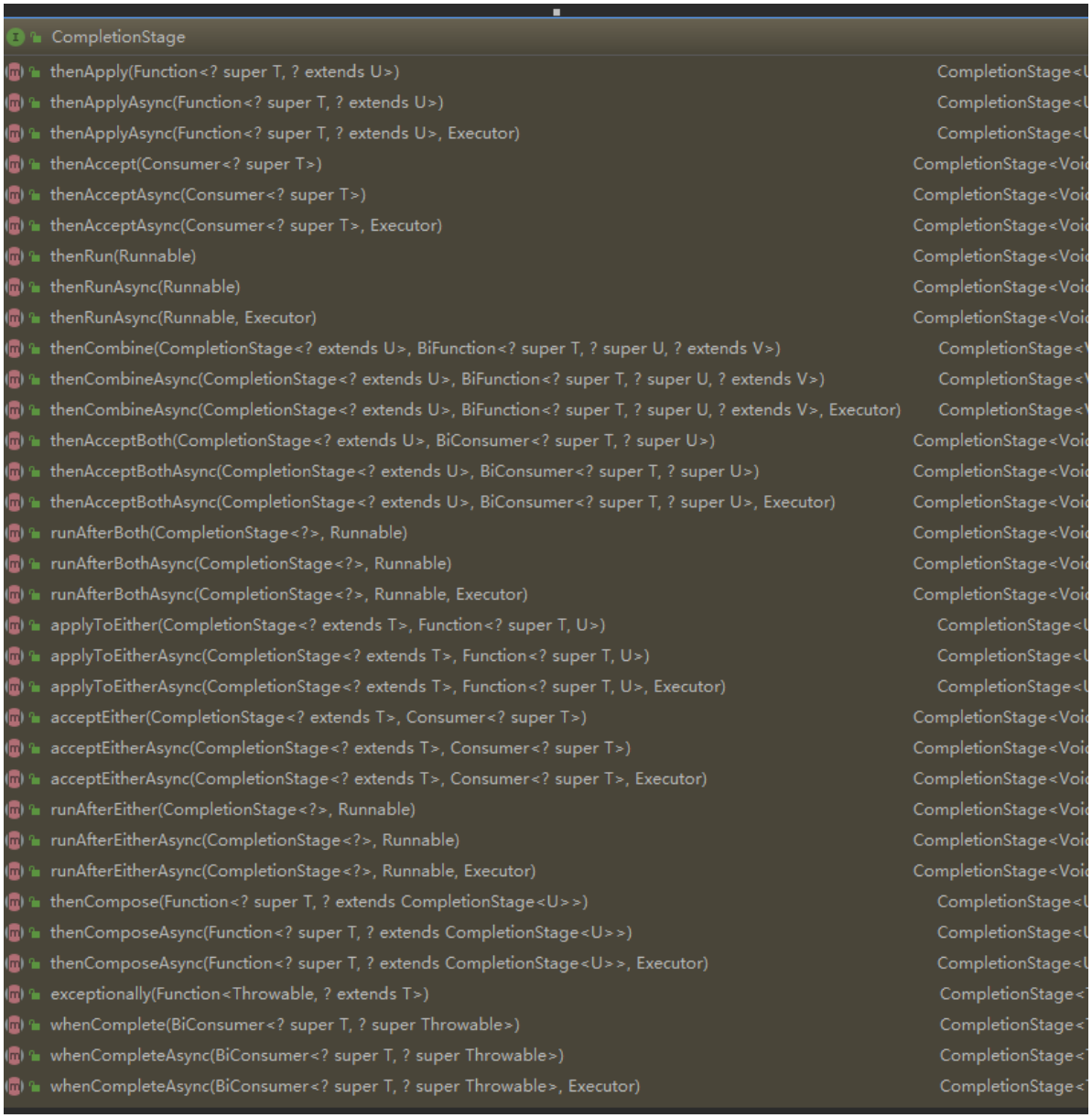
Callback方式调用，使用场景:不考虑回调时间且只能对结果做简单处理，如果依赖服务是两个或两个以上服务，则不能合并两个服务的处理结果;不阻塞主请求线程，但使用场景有限。

4、异步回调

异步回调链式编排(JDK8 CompletableFuture)，使用场景:其实不是异步调用方式，只是对依赖多服务的Callback调用结果处理做结果编排，来弥补Callback的不足，从而实现全异步链式调用。

CompletableFuture类介绍:

```
 * @author Doug Lea
 * @since 1.8
 */
public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {
 }
```



列子1： 那个线程跑的快用哪个acceptEither函数

```
1
2 public class CompletableFutureTest1{
3
4  private static CompletableFuture<Integer> m1(){
5  return CompletableFuture.supplyAsync(() -> {
6  try {
7  TimeUnit.SECONDS.sleep(1);
8  } catch (InterruptedException e) {
9  e.printStackTrace();
10  }
```

```
11  return 1111;
12  });
13  }
14  private static CompletableFuture<Integer> m2(){
15  return CompletableFuture.supplyAsync(() -> {
16  try {
17  TimeUnit.SECONDS.sleep(5);
18  } catch (InterruptedException e) {
19  e.printStackTrace();
20  }
21  return 2222;
22  });
23  }
24
25  /***
26  * acceptEither函数
27  * 那个线程跑的快用哪个
28  *
29  * @param args
30  * @throws Exception
31  */
32  public static void main(String[] args) throws Exception {
33  m1().acceptEither(m2(), t -> {
34  System.out.println("t = " + t);
35  }).get();
36  }
37 }
```

列子2：两个任何结果合并：thenCombine函数

```
1  public static void main(String[] args) {
2   try {
3   String s = CompletableFuture.supplyAsync(() -> 1)
4   .thenCombine(CompletableFuture.supplyAsync( () -> "2" ), (a, b) -> {
5   System.out.println("a =" + a);
6   System.out.println("b =" + b);
7   return a + b;
8   })
9   .get();
10   System.out.println(s);
```

```
 } catch (InterruptedException e) {
 e.printStackTrace();
 } catch (Exception e) {
 e.printStackTrace();
 }
}
```

场景：

supplyAsync thenApplyAsync

ABC C必须等到A条件完成了才完成，而B不需要等待，这种如何处理?

场景介绍：

1、获取sku的基本信息 1S

2、获取sku的图片信息 2s

3、获取sku的促销信息 3s

4、规格信息spu

5、.....

```
public static void main(String[] args) throws Exception {
 // 第一个任务:商品id
 CompletableFuture<String> cfQuery = CompletableFuture.supplyAsync(() -> {
 return query("iphone12");
 });
 // query成功后继续执行下一个任务:
 CompletableFuture<String> cfFetch = cfQuery.thenApplyAsync((code) -> {
 return comment(code);
 });
 //comment成功后打印结果:
 cfFetch.thenAccept((result) -> {
 System.out.println( result);
 });
 // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭:
 Thread.sleep(2000);
}

static String query(String name) {
 try {
 Thread.sleep(100);
 } catch (InterruptedException e) {
```

```
 }
 int random = new Random().nextInt(10);
 return random<=5?"20210709":"0000";
}

static String comment(String code) {
 if (code.equals("0000")){
 return "没有任何评论";
 }
 return "苹果手机没有华为好";

}
```

线程池优化：

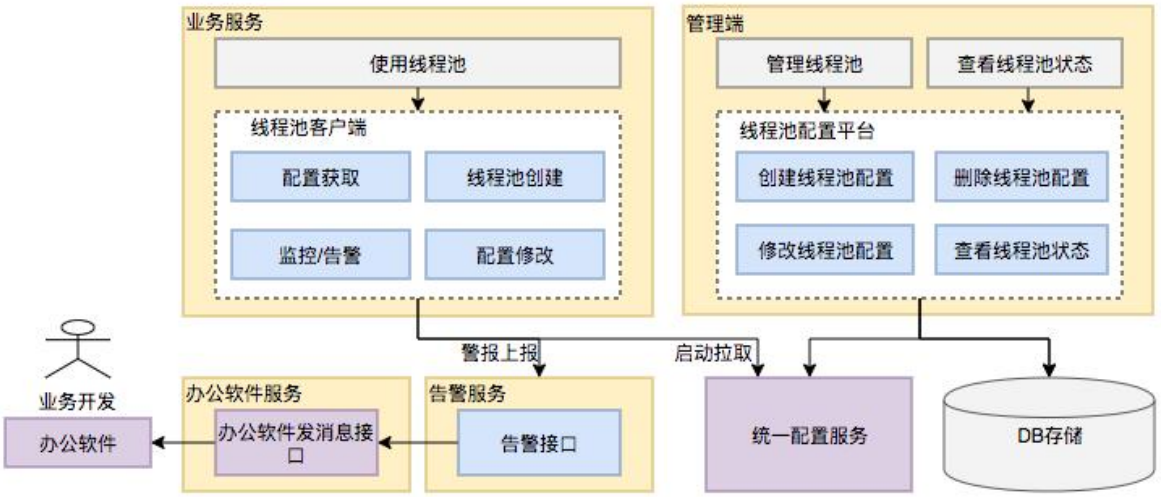
动态化线程池

动态化线程池的核心设计包括以下三个方面：

【1】简化线程池配置：线程池构造参数有8个，但是最核心的是3个：**corePoolSize**、**maximumPoolSize**，**workQueue**，它们最大程度地决定了线程池的**任务分配**和**线程分配**策略。考虑到在实际应用中我们获取并发性的场景主要是两种：（1）并行执行子任务，提高响应速度。这种情况下，应该使用**同步队列**，没有什么任务应该被缓存下来，而是应该立即执行。（2）并行执行大批次任务，提升吞吐量。这种情况下，应该使用**有界队列**，使用队列去缓冲大批量的任务，队列容量必须声明，防止任务无限制堆积。所以线程池只需要提供这三个关键参数的配置，并且提供两种队列的选择，就可以满足绝大多数的业务需求，Less is More。

【2】参数可动态修改：为了解决参数不好配，修改参数成本高等问题。在 Java线程池留有高扩展性的基础上，封装线程池，允许线程池监听同步外部的消息，根据消息进行修改配置。将线程池的配置放置在平台侧，允许开发同学简单的查看、修改线程池配置。

【3】增加线程池监控：对某事物缺乏状态的观测，就对其改进无从下手。在线程池执行任务的生命周期添加监控能力，帮助开发同学了解线程池状态。



功能架构

动态化线程池提供如下功能：

【1】动态调参：支持线程池参数动态调整、界面化操作；包括修改线程池核心大小、最大核心大小、队列长度等；参数修改后及时生效。

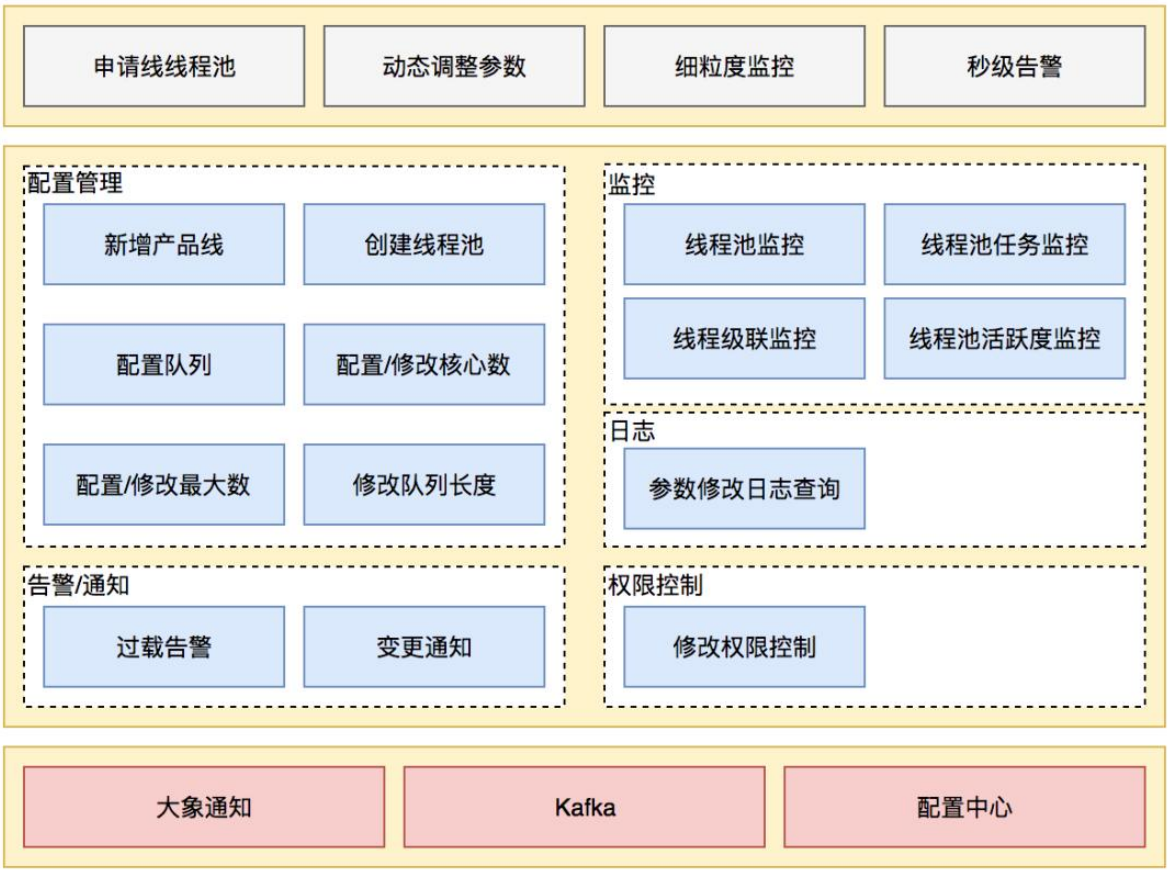
【2】任务监控：支持应用粒度、线程池粒度、任务粒度的 Transaction监控；可以看到线程池的任务执行情况、最大任务执行时间、平均任务执行时间、95/99线等。

【3】负载告警：线程池队列任务积压到一定值的时候会通过钉钉告知应用开发负责人；当线程池负载数达到一定阈值的时候会通过大象告知应用开发负责人。

【4】操作监控：创建/修改和删除线程池都会通知到应用的开发负责人。

【5】操作日志：可以查看线程池参数的修改记录，谁在什么时候修改了线程池参数、修改前的参数值是什么。

【6】权限校验：只有应用开发负责人才能够修改应用的线程池参数。



项目管理：

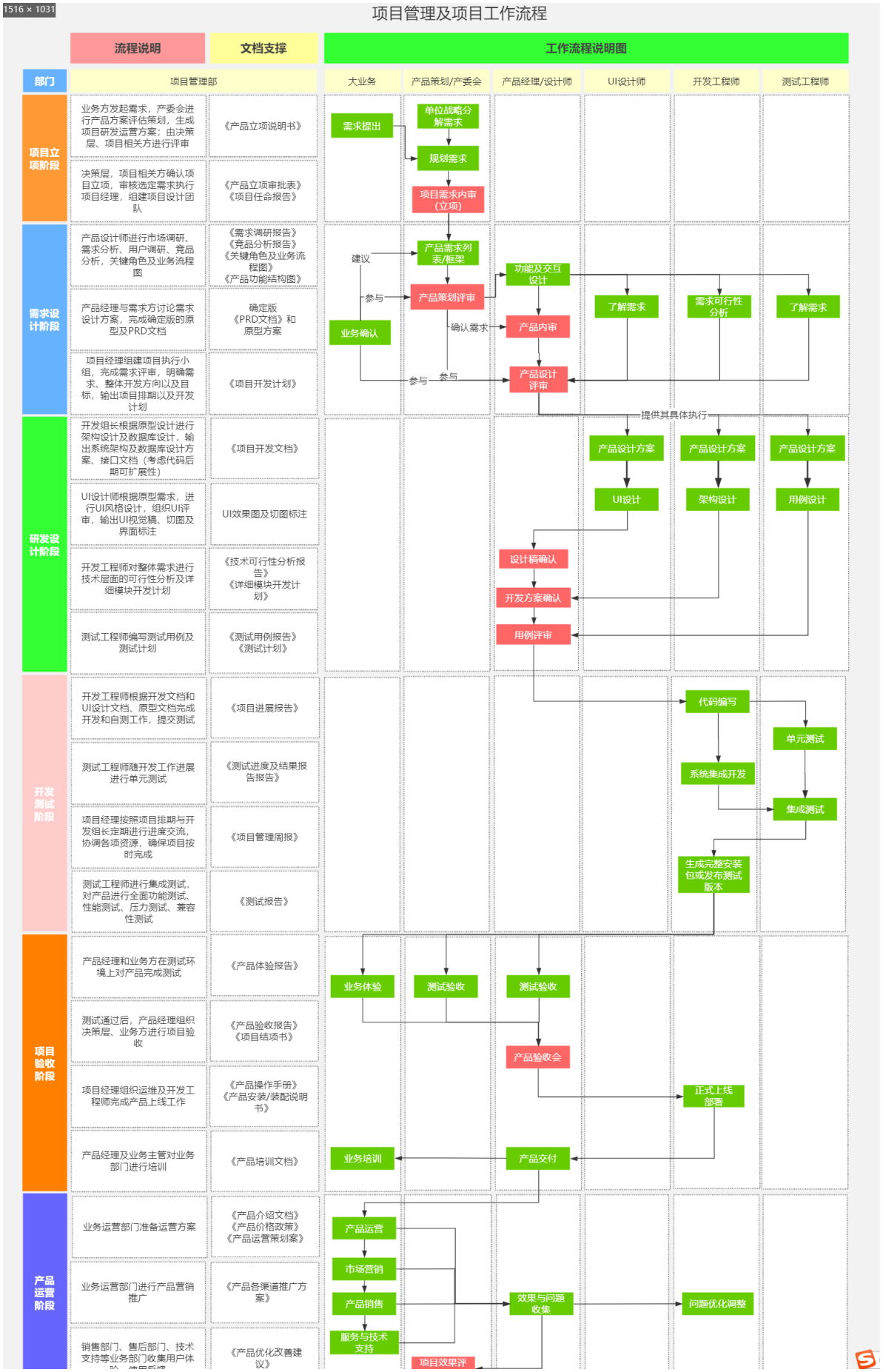
常见问题|管理误区：

为什么3天就能做完的事情？他们要一周才能做完了？
我这么努力，他们怎么不跟我一起上？
我是项目经理、为什么我说的话他们不怎么听？
为什么兄弟们不喜欢跟我开玩笑了？（好像没有以前兄弟那种感觉了）

管人、管项目

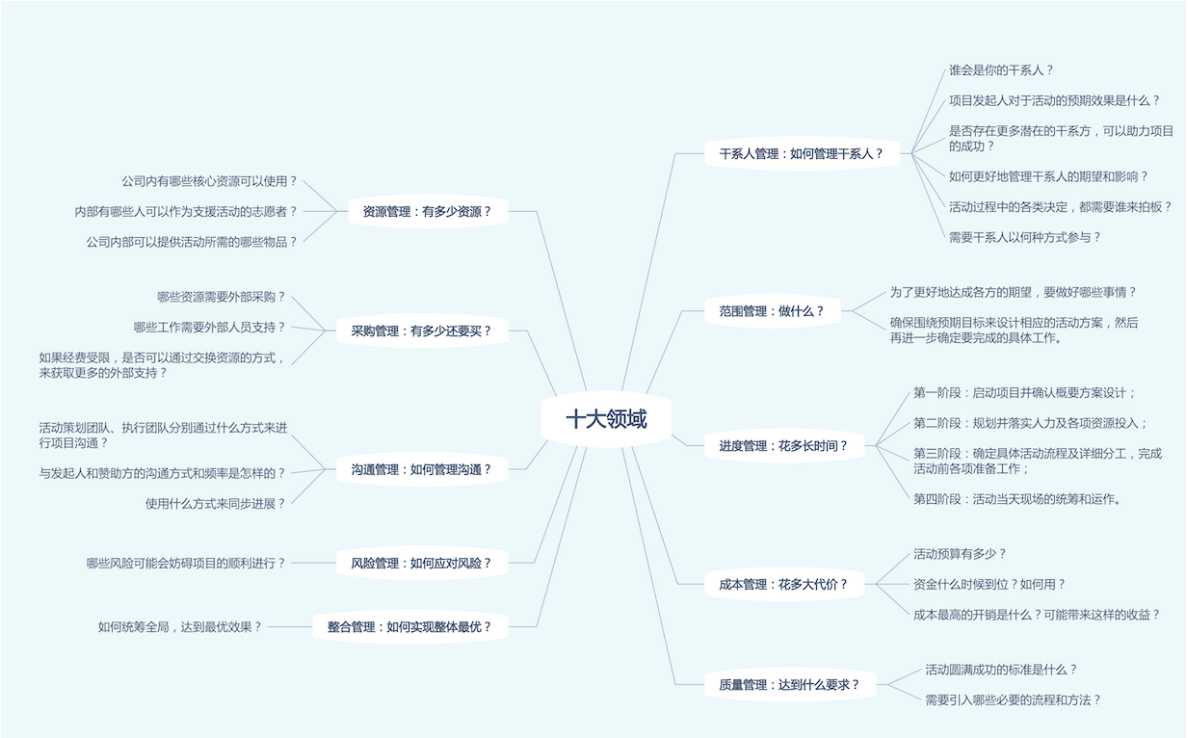
管项目：

项目进度篇



分解任务、沟通反馈、组织协调、

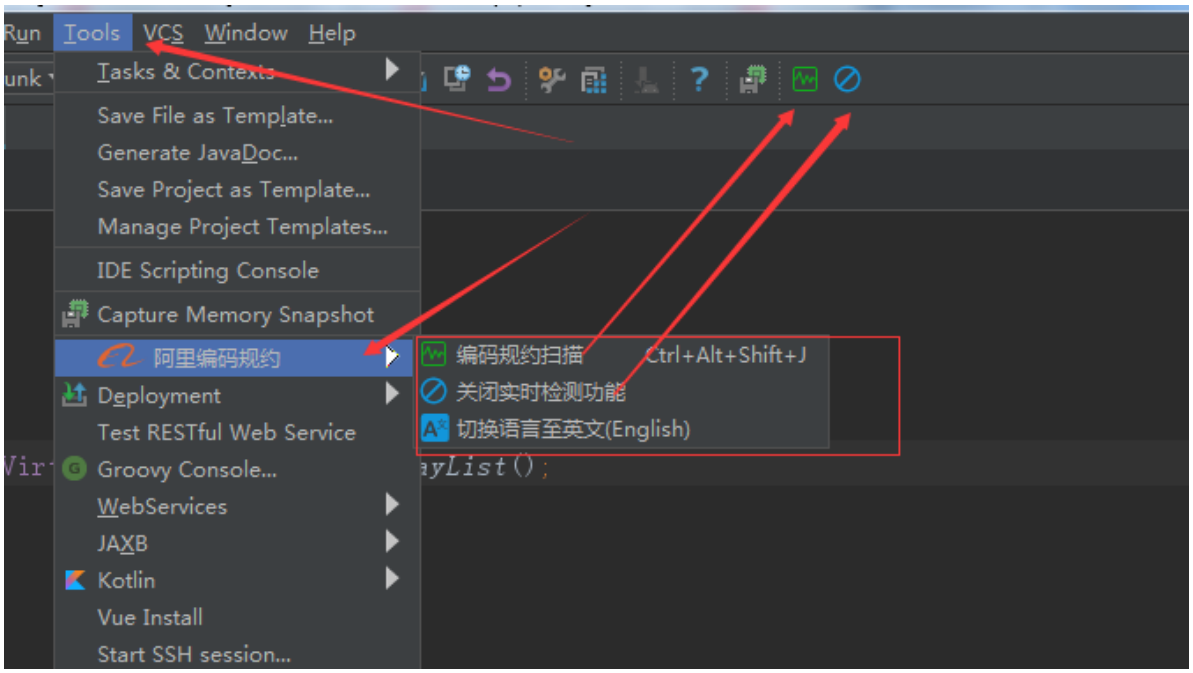
项目管理脑图:

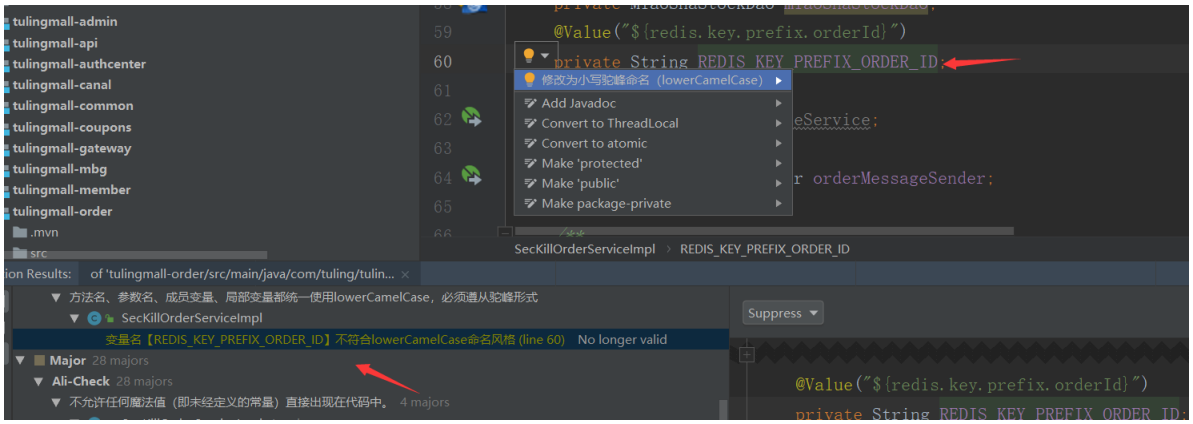


管人：

站立会、定期周会、项目奖金、亲和力、团队建设。

打开 Settings >> Plugins >> Browse repositories...





1，有可能会空指针异常，正确的写法，就是常量在前面，调用这个equals方法。

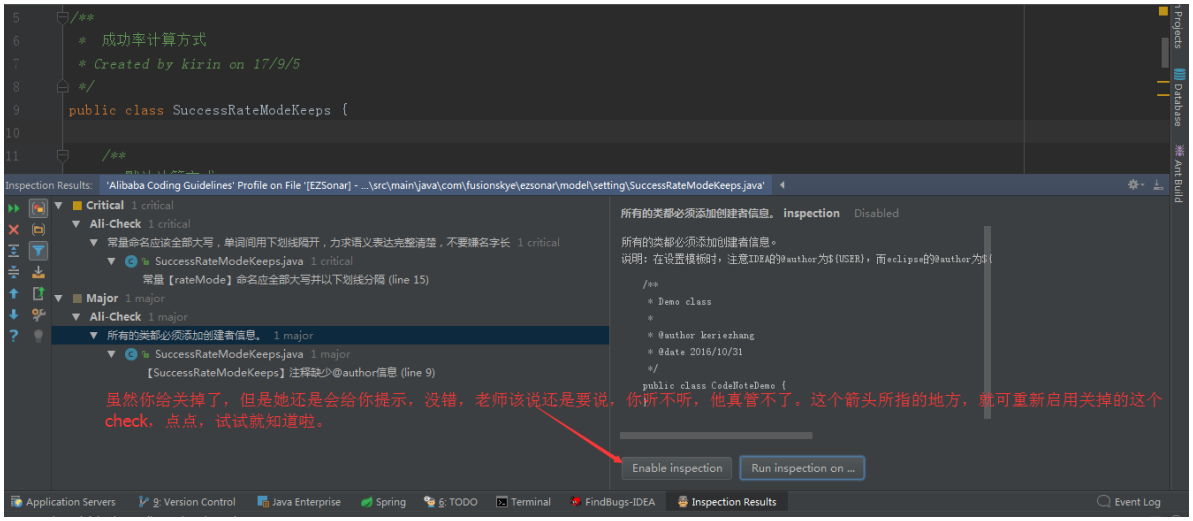
2，方法名、参数名、成员变量、局部变量都统一使用lowerCamelCase，必须遵从驼峰形式（这个也是刚刚工作的小伙伴不在意的问题）

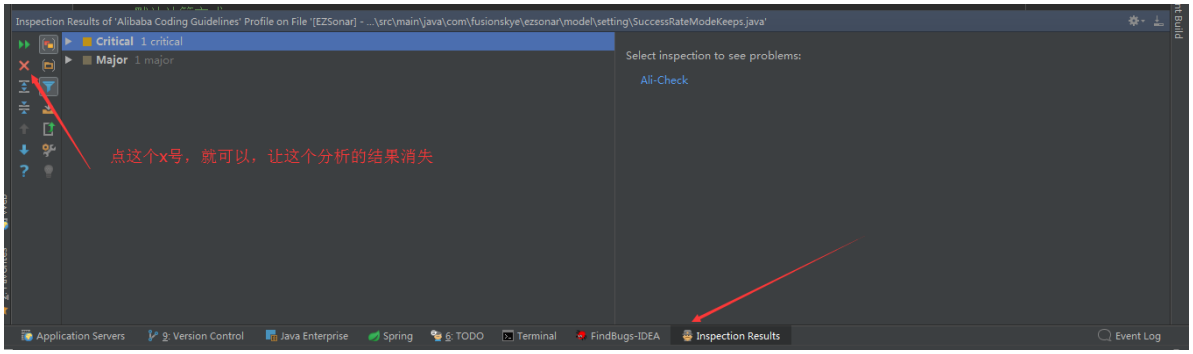
3，不允许任何魔法值（即未经定义的常量）直接出现在代码中。上面那个使用equals方法的这个字符串，是直接写在代码里面的，这个就叫做“ 魔法值 ”。

4，及时清理不再使用的代码段或配置信息。当你在改别人代码的时候，直接把不用的就删除了吧，别想着说，需求搞不好还会改回来呢。不然满屏幕都是注释的但是未删除的代码，最后，注释的代码比正儿八经工作的代码还多。

5，关于代码注释的正确书写姿势啦。这就不多说了。

关于Quick Fix 的使用(高科技，简单实用。)





文档：07项目实战总结与项目管理分享 阿里开?..

链接：http://note.youdao.com/noteshare?id=6d909ae7a3a6cada79d8ae11e0ece5d3&sub=CFC183CB5E4E4EFB8B0C6C1E8B7E8274