尚品汇商城

一、 CompletableFuture 异步编排

问题：查询商品详情页的逻辑非常复杂，数据的获取都需要远程调用，必然需要花费更多的时间。

假如商品详情页的每个查询，需要如下标注的时间才能完成

```
// 1. 获取 sku 的基本信息        1.5s

// 2. 获取 sku 的图片信息        0.5s

// 3. 获取 spu 的所有销售属性          1s

// 4. sku 价格  0.5s
...
```

那么，用户需要 3.5s 后才能看到商品详情页的内容。很显然是不能接受的。

如果有多个线程同时完成这 4 步操作，也许只需要 1.5s 即可完成响应。

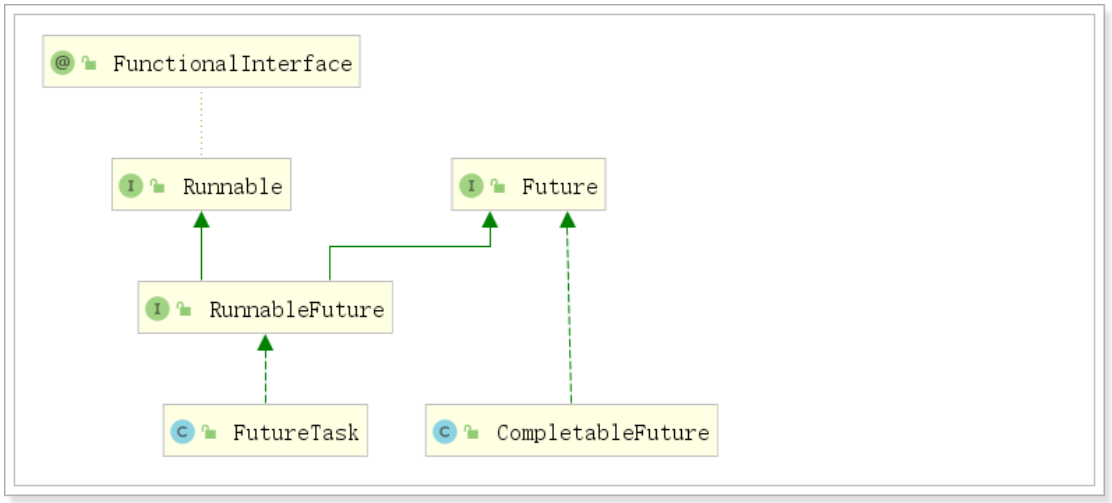
1.1 CompletableFuture 介绍

Future 是 Java 5 添加的类，用来描述一个异步计算的结果。你可以使用 isDone 方法检查计算是否完成，或者使用 get 阻塞住调用线程，直到计算完成返回结果，你也可以使用 cancel 方法停止任务的执行。

在 Java 8 中, 新增加了一个包含 50 个方法左右的类: CompletableFuture，提供了非常强大的 Future 的扩展功能，可以帮助我们简化异步编程的复杂性，提供了函数式编程的能力，可**以通过回调的方式处理计算结果**，并且提供了转换和组合 CompletableFuture 的方法。

CompletableFuture 类实现了 Future 接口，所以你还是可以像以前一样通过 get 方法阻塞或者轮询的方式获得结果，但是这种方式不推荐使用。

CompletableFuture 和 FutureTask 同属于 Future 接口的实现类，都可以获取线程的执行结果。



1.2 创建异步对象

CompletableFuture 提供了四个静态方法来创建一个异步操作。

```
public static CompletableFuture<Void>   runAsync(Runnable runnable)
public static CompletableFuture<Void>   runAsync(Runnable runnable, Executor executor)
public static <U> CompletableFuture<U>  supplyAsync(Supplier<U> supplier)
public static <U> CompletableFuture<U>  supplyAsync(Supplier<U> supplier, Executor executor)
```

没有指定 Executor 的方法会使用 **ForkJoinPool.commonPool()** 作为它的线程池执行异步代码。如果指定线程池，则　。

- runAsync 方法不支持返回值。

- supplyAsync 可以支持返回值。

1.3 计算完成时回调方法

当 CompletableFuture 的计算结果完成，或者抛出异常的时候，可以执行特定的 Action。主要是下面的方法：

```
public CompletableFuture<T>     whenComplete(BiConsumer<? super T,? super Throwable> action)
public CompletableFuture<T>     whenCompleteAsync(BiConsumer<? super T,? super Throwable> action)
public CompletableFuture<T>     whenCompleteAsync(BiConsumer<? super T,? super Throwable> action,
public CompletableFuture<T>     exceptionally(Function<Throwable,? extends T> fn)
```

whenComplete 可以处理正常或异常的计算结果，exceptionally 处理异常情况。BiConsumer<? super T,? super Throwable>可以定义处理业务

whenComplete 和 whenCompleteAsync 的区别：

whenComplete：是执行当前任务的线程执行继续执行 whenComplete 的任务。

whenComplete**Async**：是执行把 whenCompleteAsync 这个任务继续提交给线程池来进行执行。

方法不以 Async 结尾，意味着 Action 使用相同的线程执行，而 Async 可能会使用其他线程执行（如果是使用相同的线程池，也可能会被同一个线程选中执行）

代码示例：

```
public class CompletableFutureDemo {


    public static void main(String[] args) throws ExecutionException, InterruptedException {

        CompletableFuture future = CompletableFuture.supplyAsync(new Supplier<Object>() {

            @Override

            public Object get() {

                System.out.println(Thread.currentThread().getName() + "\t completableFuture");

                int i = 10 / 0;

                return 1024;

            }
```

```
        }).whenComplete(new BiConsumer<Object, Throwable>() {
            @Override
            public void accept(Object o, Throwable throwable) {
                System.out.println("-------o=" + o.toString());
                System.out.println("-------throwable=" + throwable);
            }
        }).exceptionally(new Function<Throwable, Object>() {
            @Override
            public Object apply(Throwable throwable) {
                System.out.println("throwable=" + throwable);
                return 6666;
            }
        });
        System.out.println(future.get());
    }
}
```

1.4 线程串行化与并行化方法

thenApply 方法：当一个线程依赖另一个线程时，获取上一个任务返回的结果，并返回当前任务的返回值。

```
public <U> CompletableFuture<U>     thenApply(Function<? super T,? extends U> fn)
public <U> CompletableFuture<U>     thenApplyAsync(Function<? super T,? extends U> fn)
public <U> CompletableFuture<U>     thenApplyAsync(Function<? super T,? extends U> fn, Executor e
```

thenAccept 方法：消费处理结果。接收任务的处理结果，并消费处理，无返回结果。

```
public CompletableFuture<Void>  thenAccept(Consumer<? super T> action)
public CompletableFuture<Void>  thenAcceptAsync(Consumer<? super T> action)
public CompletableFuture<Void>  thenAcceptAsync(Consumer<? super T> action, Executor executor)
```

thenRun 方法：只要上面的任务执行完成，就开始执行 thenRun，只是处理完任务后，执行 thenRun 的后续操作

```
public CompletableFuture<Void> thenRun(Runnable action) { return uniRunStage( e: null, action); }

public CompletableFuture<Void> thenRunAsync(Runnable action) { return uniRunStage(asyncPool, action); }

public CompletableFuture<Void> thenRunAsync(Runnable action,
                                            Executor executor) {
    return uniRunStage(screenExecutor(executor), action);
}
```

带有 Async 默认是异步执行的。这里所谓的异步指的是不在当前线程内执行。

Function<? super T,? extends U>

T：上一个任务返回结果的类型

U：当前任务的返回值类型

代码演示：

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    CompletableFuture<Integer> future = CompletableFuture.supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            System.out.println(Thread.currentThread().getName() + "\t completableFuture");
            //int i = 10 / 0;
            return 1024;
        }
    }).thenApply(new Function<Integer, Integer>() {
        @Override
        public Integer apply(Integer o) {
            System.out.println("thenApply 方法，上次返回结果：" + o);
            return  o * 2;
        }
    }).whenComplete(new BiConsumer<Integer, Throwable>() {
        @Override
```

```
        public void accept(Integer o, Throwable throwable) {
            System.out.println("-------o=" + o);
            System.out.println("-------throwable=" + throwable);
        }
    }).exceptionally(new Function<Throwable, Integer>() {
        @Override
        public Integer apply(Throwable throwable) {
            System.out.println("throwable=" + throwable);
            return 6666;
        }
    });
    System.out.println(future.get());
}
```

并行化：

```
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(50, 500, 30, TimeUnit.SECONDS, new ArrayBlockingQueue<>(10000));


// 线程1 执行返回的结果：hello
CompletableFuture<String> futureA = CompletableFuture.supplyAsync(() -> "hello");


// 线程2 获取到线程1 执行的结果
CompletableFuture<Void> futureB = futureA.thenAcceptAsync((s) -> {
    delaySec(1);
    printCurrTime(s+" 第一个线程");
}, threadPoolExecutor);
```

```
CompletableFuture<Void> futureC = futureA.thenAcceptAsync((s)
-> {
    delaySec(3);
    printCurrTime(s+" 第二个线程");
}, threadPoolExecutor);

private static void printCurrTime(String str) {
    System.out.println(str);
}

private static void delaySec(int i) {
    try {
        Thread.sleep(i*1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

测试：改变线程的睡眠时间，则会交替打印，则说明是并行执行.

1.5 多任务组合

```
public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs);
public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs);
```

allOf：等待所有任务完成

anyOf：只要有一个任务完成

1.6 优化商品详情页

```
@Service
public class ItemServiceImpl implements ItemService {


    @Autowired
    private ProductFeignClient productFeignClient;



    @Autowired
    private ThreadPoolExecutor threadPoolExecutor;


    @Override
    public Map<String, Object> getBySkuId(Long skuId) {


        Map<String, Object> result = new HashMap<>();


        // 通过skuId 查询skuInfo
        CompletableFuture<SkuInfo>       skuCompletableFuture       =
CompletableFuture.supplyAsync(() -> {
            SkuInfo skuInfo = productFeignClient.getSkuInfo(skuId);
            // 保存skuInfo
            result.put("skuInfo", skuInfo);
            return skuInfo;
        }, threadPoolExecutor);



        // 销售属性-销售属性值回显并锁定
    CompletableFuture<Void>      spuSaleAttrCompletableFuture      =
skuCompletableFuture.thenAcceptAsync(skuInfo -> {
            List<SpuSaleAttr>            spuSaleAttrList            =
```

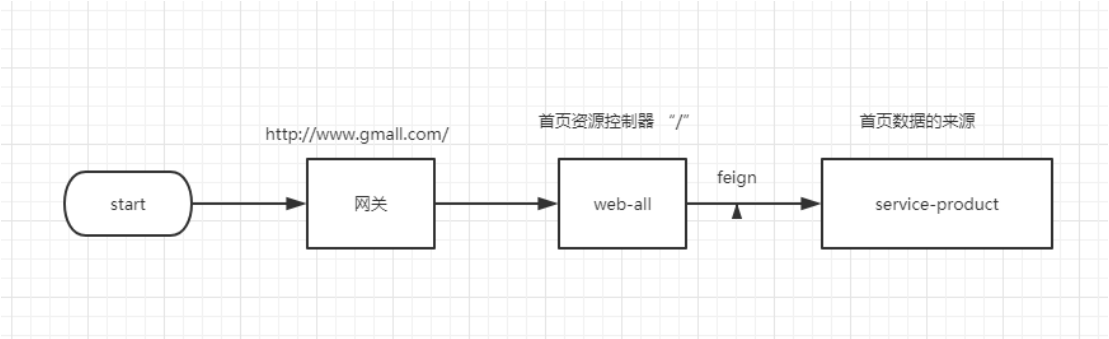
```
productFeignClient.getSpuSaleAttrListCheckBySku(skuInfo.getId(),
skuInfo.getSpuId());


            // 保存数据

      result.put("spuSaleAttrList", spuSaleAttrList);

        }, threadPoolExecutor);


        //根据 spuId 查询 map 集合属性

    // 销售属性-销售属性值回显并锁定

    CompletableFuture<Void>     skuValueIdsMapCompletableFuture     =
skuCompletableFuture.thenAcceptAsync(skuInfo -> {

            Map                     skuValueIdsMap                  =
productFeignClient.getSkuValueIdsMap(skuInfo.getSpuId());


            String                    valuesSkuJson                 =
JSON.toJSONString(skuValueIdsMap);

            // 保存 valuesSkuJson

            result.put("valuesSkuJson", valuesSkuJson);

        }, threadPoolExecutor);



        //获取商品最新价格

    CompletableFuture<Void>        skuPriceCompletableFuture        =
CompletableFuture.runAsync(() -> {

            BigDecimal                    skuPrice                  =
productFeignClient.getSkuPrice(skuId);

            result.put("price", skuPrice);

        }, threadPoolExecutor);



        //获取分类信息

    CompletableFuture<Void>      categoryViewCompletableFuture      =
skuCompletableFuture.thenAcceptAsync(skuInfo -> {
```

```
            BaseCategoryView categoryView = productFeignClient.getCategoryView(skuInfo.getCategory3Id());


            //分类信息
        result.put("categoryView", categoryView);
          }, threadPoolExecutor);


//  获取海报数据
CompletableFuture<Void> spuPosterListCompletableFuture = skuInfoCompletableFuture.thenAcceptAsync(skuInfo -> {
    //  spu 海报数据
    List<SpuPoster> spuPosterList = productFeignClient.findSpuPosterBySpuId(skuInfo.getSpuId());
    result.put("spuPosterList", spuPosterList);
},threadPoolExecutor);


//  获取 sku 平台属性，即规格数据
CompletableFuture<Void> skuAttrListCompletableFuture = CompletableFuture.runAsync(() -> {
    List<BaseAttrInfo> attrList = productFeignClient.getAttrList(skuId);
    //  使用拉姆达表示
    List<Map<String, String>> skuAttrList = attrList.stream().map((baseAttrInfo) -> {
        Map<String, String> attrMap = new HashMap<>();
        attrMap.put("attrName", baseAttrInfo.getAttrName());
        attrMap.put("attrValue", baseAttrInfo.getAttrValueList().get(0).getValueName());
        return attrMap;
    }).collect(Collectors.toList());
    result.put("skuAttrList", skuAttrList);
},threadPoolExecutor);
```

```
        CompletableFuture.allOf(skuCompletableFuture,
spuSaleAttrCompletableFuture,
skuValueIdsMapCompletableFuture,skuPriceCompletableFuture,
categoryViewCompletableFuture,spuPosterListCompletableFuture,skuAttrLis
tCompletableFuture ).join();

        return result;

    }

}
```

```
package com.atguigu.gmall.item.config;

@Configuration
public class ThreadPoolConfig {
    @Bean
    public ThreadPoolExecutor threadPoolExecutor(){
        /**
         * 核心线程数
    * 拥有最多线程数
    * 表示空闲线程的存活时间
    * 存活时间单位
    * 用于缓存任务的阻塞队列
    * 省略:
    *  threadFactory：指定创建线程的工厂
    *  handler：表示当 workQueue 已满，且池中的线程数达到 maximumPoolSize 时，线
程池拒绝添加新任务时采取的策略。
    */
        return   new   ThreadPoolExecutor(50,500,30,   TimeUnit.SECONDS,new
ArrayBlockingQueue<>(10000));
    }
}
```

二、首页商品分类实现



前面做了商品详情，我们现在来做首页分类，我先看看京东的首页分类效果，我们如何实现类似效果：

思路：

1，首页属于并发量比较高的访问页面，我看可以采取页面静态化方式实现，或者把数据放在缓存中实现

2，我们把生成的静态文件可以放在 nginx 访问或者放在 web-all 模块访问

2.1 修改 web-all 模块

2.1.1 修改 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project                    xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent>

        <groupId>com.atguigu.gmall</groupId>

        <artifactId>web</artifactId>

        <version>1.0</version>

    </parent>


    <artifactId>web-all</artifactId>

    <version>1.0</version>


    <packaging>jar</packaging>

    <name>web-index</name>

    <description>web-all</description>


    <dependencies>

     <dependency>

            <groupId>com.atguigu.gmall</groupId>

            <artifactId>service-item-client</artifactId>

            <version>1.0</version>

        </dependency>
```

```
        <dependency>
            <groupId>com.atguigu.gmall</groupId>
            <artifactId>service-product-client</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>


</project>
```

2.2 封装数据接口

由于商品分类信息在 service-product 模块，我们在该模块封装数据，数据结构为父子层级

商品分类保存在 base_category1、base_category2 和 base_category3 表中，由于需要静态化页面，我们需要一次性加载所有数据，前面我们使用了一个视图 base_category_view，所有我从视图里面获取数据，然后封装为父子层级

数据结构如下：json 数据结构

```
[
  {
    "index": 1,
    "categoryChild": [
      {
        "categoryChild": [
          {
            "categoryName": "电子书", # 三级分类的 name
```

```
            "categoryId": 1
          },
          {
            "categoryName": "网络原创", # 三级分类的 name
            "categoryId": 2
          },
          ...
        ],
        "categoryName": "电子书刊", #二级分类的 name
        "categoryId": 1
      },
     ...
    ],
    "categoryName": "图书、音像、电子书刊", # 一级分类的 name
    "categoryId": 1
  },
  ...
  "index": 2,
      "categoryChild": [
        {
          "categoryChild": [
            {
              "categoryName": "超薄电视", # 三级分类的 name
              "categoryId": 1
            },
            {
              "categoryName": "全面屏电视", # 三级分类的 name
              "categoryId": 2
            },
            ...
```

```
            ],
            "categoryName": "电视", #二级分类的 name
            "categoryId": 1
          },
         ...
        ],
        "categoryName": "家用电器", # 一级分类的 name
        "categoryId": 2
    }
]
```

2.2.1 ManageService 接口

```
/**
 * 获取全部分类信息
 * @return
 */
List<JSONObject> getBaseCategoryList();
```

2.2.2 ManageServiceImpl 实现类

```
@Override
@GmallCache(prefix = "category")
public List<JSONObject> getBaseCategoryList() {
    // 声明几个json 集合
  ArrayList<JSONObject> list = new ArrayList<>();
    // 声明获取所有分类数据集合
  List<BaseCategoryView>                baseCategoryViewList                =
baseCategoryViewMapper.selectList(null);
    // 循环上面的集合并安一级分类 Id 进行分组
  Map<Long,       List<BaseCategoryView>>       category1Map                =
baseCategoryViewList.stream().collect(Collectors.groupingBy(BaseCategoryVie
w::getCategory1Id));
    int index = 1;
```

```
    // 获取一级分类下所有数据
  for     (Map.Entry<Long,     List<BaseCategoryView>>     entry1          :
category1Map.entrySet()) {
        // 获取一级分类 Id
        Long category1Id  = entry1.getKey();
        // 获取一级分类下面的所有集合
   List<BaseCategoryView> category2List1  = entry1.getValue();
        //
        JSONObject category1 = new JSONObject();
        category1.put("index", index);
        category1.put("categoryId",category1Id);
        // 一级分类名称
   category1.put("categoryName",category2List1.get(0).getCategory1Name());
        // 变量迭代
   index++;
        // 循环获取二级分类数据
   Map<Long,       List<BaseCategoryView>>       category2Map              =
category2List1.stream().collect(Collectors.groupingBy(BaseCategoryView::get
Category2Id));
        // 声明二级分类对象集合
   List<JSONObject> category2Child = new ArrayList<>();
        // 循环遍历
   for     (Map.Entry<Long,     List<BaseCategoryView>>     entry2          :
category2Map.entrySet()) {
            // 获取二级分类 Id
            Long category2Id  = entry2.getKey();
            // 获取二级分类下的所有集合
     List<BaseCategoryView> category3List  = entry2.getValue();
            // 声明二级分类对象
     JSONObject category2 = new JSONObject();

            category2.put("categoryId",category2Id);

category2.put("categoryName",category3List.get(0).getCategory2Name());
            // 添加到二级分类集合
     category2Child.add(category2);

            List<JSONObject> category3Child = new ArrayList<>();

            // 循环三级分类数据
     category3List.stream().forEach(category3View -> {
                JSONObject category3 = new JSONObject();
                category3.put("categoryId",category3View.getCategory3Id());

category3.put("categoryName",category3View.getCategory3Name());

                category3Child.add(category3);
            });

            // 将三级数据放入二级里面
     category2.put("categoryChild",category3Child);

        }
        // 将二级数据放入一级里面
   category1.put("categoryChild",category2Child);
```

```
            list.add(category1);
        }
        return list;
    }
```

2.2.3 控制器

ProductApiController

```
/**
 * 获取全部分类信息
 * @return
 */
@GetMapping("getBaseCategoryList")
public Result getBaseCategoryList(){
    List<JSONObject> list = manageService.getBaseCategoryList();
    return Result.ok(list);
}
```

2.3 service-product-client 添加接口

ProductFeignClient

```
/**
 * 获取全部分类信息
 * @return
 */
@GetMapping("/api/product/getBaseCategoryList")
Result getBaseCategoryList();
```

ProductDegradeFeignClient

```
@Override
public Result getBaseCategoryList() {
    return null;
}
```

2.4 页面渲染

第一种缓存渲染方式：

```
package com.atguigu.gmall.all.controller;


@Controller
public class IndexController {


    @Autowired
    private ProductFeignClient productFeignClient;


    @GetMapping({"/","index.html"})
    public String index(HttpServletRequest request){
      // 获取首页分类数据
   Result result = productFeignClient.getBaseCategoryList();
      request.setAttribute("list",result.getData());
      return "index/index";
  }
}
```

第二种 nginx 做静态代理方式：

```
@GetMapping("createIndex")
@ResponseBody
public Result createIndex(){
    //  获取后台存储的数据
    Result result = productFeignClient.getBaseCategoryList();
    //  设置模板显示的内容
    Context context = new Context();
    context.setVariable("list",result.getData());

    //  定义文件输入位置
    FileWriter fileWriter = null;
    try {
        fileWriter = new FileWriter("D:\\index.html");
    } catch (IOException e) {
        e.printStackTrace();
    }
    //  调用process();方法创建模板
    templateEngine.process("index/index",context,fileWriter);
    return Result.ok();
}
```

将生成的静态页面与 css 放入 html 中即可!