## 一.业务背景

1.神农口袋随着业务的发展和用户量，数据量的增长，带来的现有接口的同步调用耗时越来越久，例如归档接口 “归档接口截图” 结论：大致思路是将同步调用改造成异步调用来优化接口性能

2.为什么选用CompletableFuture 对业务流程进行编排，降低依赖之间的阻塞来减少线程池的调度开销和阻塞时间； 2.1 Future接口的局限性 所谓异步调用其实就是实现一个可无需等待被调用函数的返回值而让操作继续运行的方法。 在 Java 语言中，简单的讲就是另启一个线程来完成调用中的部分计算，使调用继续运行或返回，而不需要等待计算结果。但调用者仍需要取线程的计算结果 DK5新增了Future接口，用于描述一个异步计算的结果。虽然 Future 以及相使关用方法提供了异步执行任务的能力，但是对于结果的获取却是很不方便， 只能通过阻塞或者轮询的方式得到任务的结果。阻塞的方式显然和我们的异步编程的初衷相违背，轮询的方式又会耗费无谓的 CPU 资源，而且也不能及时地得到计算结果, 没有异常处理； **一. Future**

JDK 5引入了Future模式。Future接口是Java多线程Future模式的实现，在java.util.concurrent包中，可以来进行异步计算。

Future模式是多线程设计常用的一种设计模式。Future模式可以理解成：我有一个任务，提交给了Future，Future替我完成这个任务。期间我自己可以去做任何想做的事情。一段时间之后，我就便可以从Future那儿取出结果。

Future的接口很简单，只有五个方法。

public interface Future<V> {  
  
 boolean cancel(boolean mayInterruptIfRunning);  
  
 boolean isCancelled();  
  
 boolean isDone();  
  
 V get() throws InterruptedException, ExecutionException;  
  
 V get(long timeout, TimeUnit unit)  
 throws InterruptedException, ExecutionException, TimeoutException;  
}

Future接口的方法介绍如下：

* boolean cancel (boolean mayInterruptIfRunning) 取消任务的执行。参数指定是否立即中断任务执行，或者等等任务结束
* boolean isCancelled () 任务是否已经取消，任务正常完成前将其取消，则返回 true
* boolean isDone () 任务是否已经完成。需要注意的是如果任务正常终止、异常或取消，都将返回true
* V get () throws InterruptedException, ExecutionException 等待任务执行结束，然后获得V类型的结果。InterruptedException 线程被中断异常， ExecutionException任务执行异常，如果任务被取消，还会抛出CancellationException
* V get (long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException 同上面的get功能一样，多了设置超时时间。参数timeout指定超时时间，uint指定时间的单位，在枚举类TimeUnit中有相关的定义。如果计 算超时，将抛出TimeoutException

一般情况下，我们会结合Callable和Future一起使用，通过ExecutorService的submit方法执行Callable，并返回Future。

ExecutorService executor = Executors.newCachedThreadPool();  
  
 Future<String> future = executor.submit(() -> { //Lambda 是一个 callable， 提交后便立即执行，这里返回的是 FutureTask 实例  
 System.out.println("running task");  
 Thread.sleep(10000);  
 return "return task";  
 });  
  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 }  
  
 System.out.println("do something else"); //前面的的 Callable 在其他线程中运行着，可以做一些其他的事情  
  
 try {  
 System.out.println(future.get()); //等待 future 的执行结果，执行完毕之后打印出来  
 } catch (InterruptedException e) {  
 } catch (ExecutionException e) {  
  
 } finally {  
 executor.shutdown();  
 }

比起future.get()，其实更推荐使用get (long timeout, TimeUnit unit) 方法，设置了超时时间可以防止程序无限制的等待future的结果。

Future接口可以构建异步应用，但依然有其局限性。它很难直接表述多个Future 结果之间的依赖性。实际开发中，我们经常需要达成以下目的：

1. 将多个异步计算的结果合并成一个
2. 等待Future集合中的所有任务都完成
3. Future完成事件（即，任务完成以后触发执行动作）

2.2 CompletableFuture的优点

1. CompletableFuture支持手动完成任务，比如被调用的远程服务无响应，可以手动返回一个结果；Future则会导致调用方挂起。
2. CompletableFuture支持callback，Future只能通过get阻塞的获取结果。
3. CompletableFuture支持组合Future，更加方便编排异步请求

使用CompletableFuture优雅的处理不通业务场景

业务场景可以大致划分为2类场景，Fork调用，Join调用

1. Fork类型 接口CF1成功后调用CF2、CF3；
2. Join类型 接口CF3依赖接口CF1、CF2同时成功后再执行

根据CompletableFuture依赖数量，可以将上述场景的CompletableFuture的创建分为以下几类：零源依赖、一元依赖、二元依赖和多元依赖。

* 可组合：可以将多个依赖操作通过不同的方式进行编排，例如CompletableFuture提供thenCompose、thenCombine等各种then开头的方法，这些方法就是对“可组合”特性的支持。
* 操作融合：将数据流中使用的多个操作符以某种方式结合起来，进而降低开销（时间、内存）。
* 延迟执行：操作不会立即执行，当收到明确指示时操作才会触发。例如Reactor只有当有订阅者订阅时，才会触发操作。
* 回压：某些异步阶段的处理速度跟不上，直接失败会导致大量数据的丢失，对业务来说是不能接受的，这时需要反馈上游生产者降低调用量。

## 常见非阻塞编程“模型”对比

* **Composable - 可组合** ：可以将多个依赖操作通过不同的方式进行编排，例如CompletableFuture提供thenCompose、thenCombine等各种then开头的方法，这些方法就是对“可组合”特性的支持。
* **Operator fusion - 操作融合**：将数据流中使用的多个操作符以某种方式结合起来，如just、from、flatMap等
* **Lazy - 延迟执行**：操作不会立即执行，当收到明确指示时操作才会触发。例如Reactor只有当有订阅者订阅时，才会触发操作。
* **Backpressure - 背压**：将压力返回到调用者，这样调用方可以感知下游压力

## 二.实践探究

### 1.常用api介绍

#### 1.1 创建异步任务（runAsync，supplyAsync）

//runAsync方法不支持返回值  
public static CompletableFuture<Void> runAsync(Runnable runnable)  
public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)  
//supplyAsync可以支持返回值  
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)  
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)

#### 1.2 获得结果和触发计算(get、getNow、join、complete)

public T get( ) //不见不散(会抛出异常) 只要调用了get( )方法,不管是否计算完成都会导致阻塞   
public T get(long timeout, TimeUnit unit) //过时不候   
public T getNow(T valuelfAbsent) //没有计算完成的情况下,给我一个替代结果计算完,返回计算完成后的结果、没算完,返回设定的valuelfAbsent   
public T join( ) //join方法和get( )方法作用一样,不同的是,join方法不抛出异常   
public boolean complete(T value) //主动结束异步任务，可以打断get方法，立即返回括号值

#### 1.3 对计算结果进行消费(thenRun、thenAccept、thenApply)

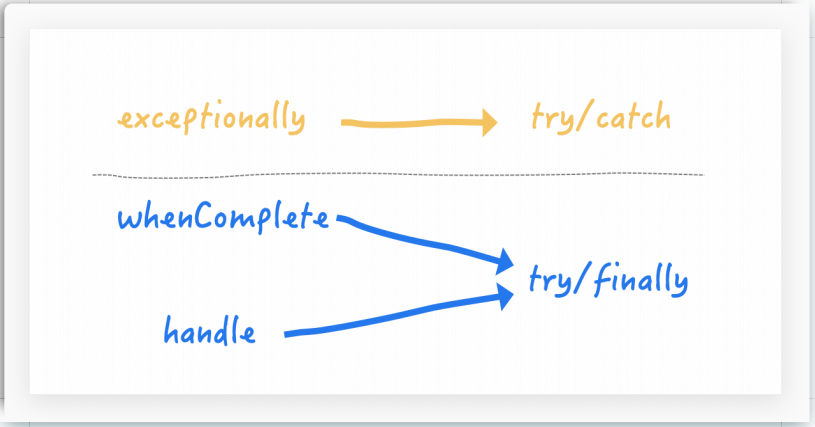
都有对应的异步方法

thenRun(Runnable runnable) //无参无返  
CompletableFuture<Void> thenAccept(Consumer<? super T> action) //消费型，有参无返  
public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn) //函数型，有参有返

#### 1.4 对计算结果进行处理(exceptionally、whenComplete、handle)

都有对应的异步方法

CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn) //就相当于 catch,出现异常,将会跳过 thenApply 的后续操作,直接捕获异常，进行一场处理  
CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable> action) //有异常也可以往下一步走,根据带的异常参数可以进一步处理,无返回值   
<U> CompletableFuture<U> handle(BiFunction<? super T, Throwable, ? extends U> fn) //有异常也可以往下一步走,根据带的异常参数可以进一步处理,有返回值

对比try/catch/finally 

#### 1.5 对计算速度进行选用(applyToEither、acceptEither、runAfterEither)

都有对应的异步方法

public <U> CompletableFuture<U> applyToEither(CompletionStage<? extends T> other, Function<? super T, U> fn) //这个方法表示的是,2个异步任务谁快就用谁的结果  
  
//测试  
public void test() {  
 System.out.println(CompletableFuture.supplyAsync(() -> {  
 //暂停几秒钟线程  
 try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) {e.printStackTrace();}  
 return 1;  
 }).applyToEither(CompletableFuture.supplyAsync(() -> {  
 try { TimeUnit.SECONDS.sleep(2);} catch (InterruptedException e) { e.printStackTrace();}  
 return 2;  
 }),r -> r).join());  
 //主线程暂停几秒钟线程防止退出  
 try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException e) { e.printStackTrace(); }  
}

#### 1.6 对计算结果进行合并(thenCombine、thenAcceptBoth、runAfterBoth)

都有对应的异步方法

public <U,V> CompletableFuture<V> thenCombine(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn)//两个CompletionStage任务都完成后,最终把两个任务的结果一起交给thenCombine来处理,先完成的先等着,等待其他分支任务  
public <U> CompletableFuture<Void> thenAcceptBoth(CompletionStage<? extends U> other,BiConsumer<? super T, ? super U> action) //与thenCombine相比是没有返回参数  
public CompletableFuture<Void> runAfterBoth(CompletionStage<?> other,Runnable action) //无参无返  
  
//测试  
public void test() {  
System.out.println(CompletableFuture.supplyAsync(() -> 10).thenCombine(CompletableFuture.supplyAsync(() -> 20), (r1, r2) -> r1 + r2).thenCombine(CompletableFuture.supplyAsync(() -> 30), (r3, r4) -> r3 + r4).join());  
}

#### 1.7 多任务组合(allOf、anyOf)

public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) //allOf:等待所有任务完成  
public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs) //anyOf:只要有一个任务完成

### 2.案例分析（注销登记接口）

时序图

代码一：  代码二： 

### 3.问题归纳

#### 3.1 代码执行在哪个线程上？

多线程编程的前提是要清楚自己写的每一行代码会在哪一个线程上执行，CompletableFuture实现了CompletionStage接口，通过丰富的回调方法，实现了各种组合操作，每种组合场景都有同步和异步两种方法。

##### 3.1.1同步方法（不带Async后缀的方法）的代码执行有两种情况。

* 如果注册时被依赖的操作已经执行完成，则直接由当前线程(main线程)执行。（异步计算只会计算一次）
* 如果注册时被依赖的操作还未执行完，则由回调线程执行。

##### 3.1.2异步方法（带Async后缀的方法）

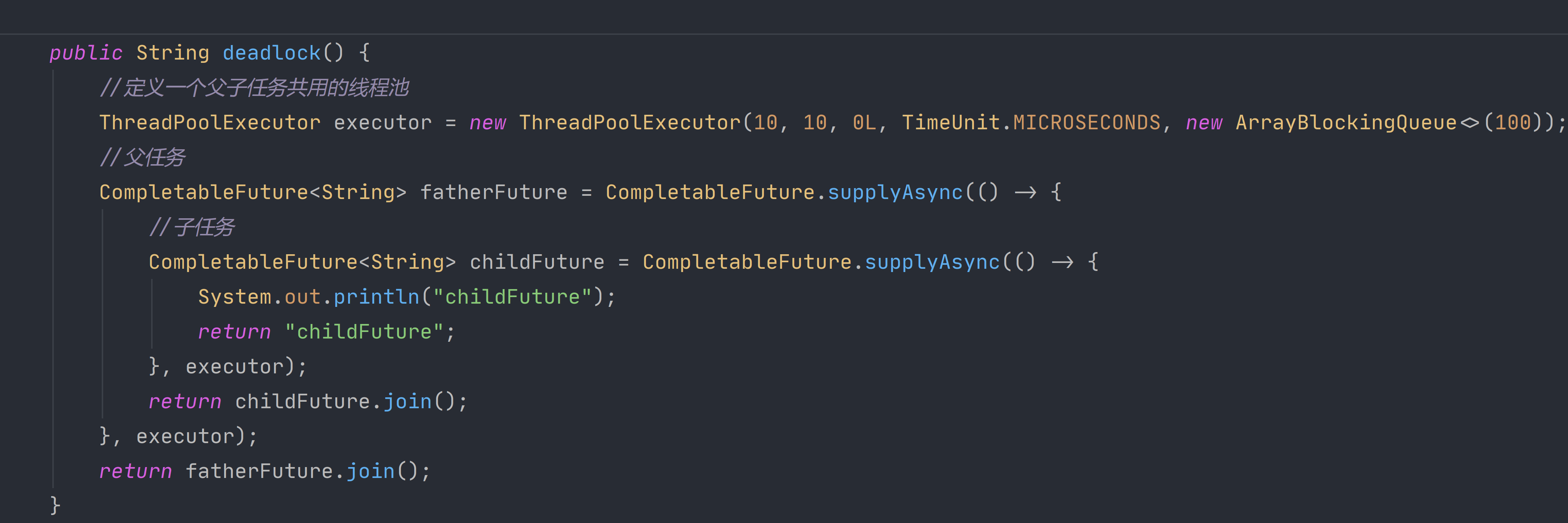
* 可以选择是否传递线程池参数Executor运行在指定线程池中；当不传递Executor时，会使用ForkJoinPool中的共用线程池CommonPool（CommonPool的大小是CPU核数-1，如果是IO密集的应用，线程数可能成为瓶颈）。

#### 3.2 线程池相关问题

##### 3.2.1 异步回调要传线程池

前面提到，异步回调方法可以选择是否传递线程池参数Executor，一般建议**强制传线程池，且根据实际情况做线程池隔离**。 当不传递线程池时，会使用ForkJoinPool中的公共线程池CommonPool，这里所有调用将共用该线程池，核心线程数=处理器数量-1（单核核心线程数为1），所有异步回调都会共用该CommonPool，核心与非核心业务都竞争同一个池中的线程，很容易成为系统瓶颈。手动传递线程池参数可以更方便的调节参数，并且可以给不同的业务分配不同的线程池，以求资源隔离，减少不同业务之间的相互干扰。

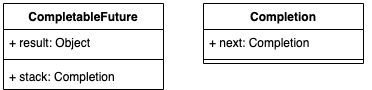
##### 3.2.2 循环引用造成死锁

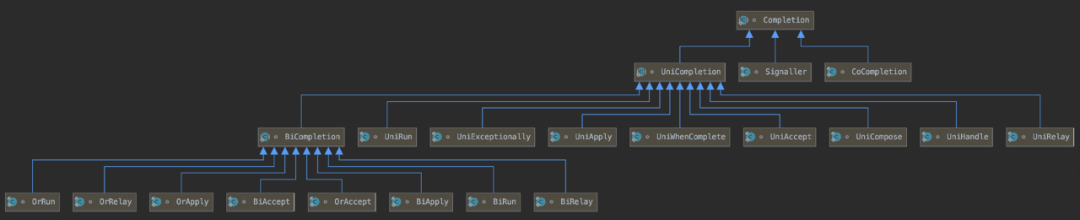
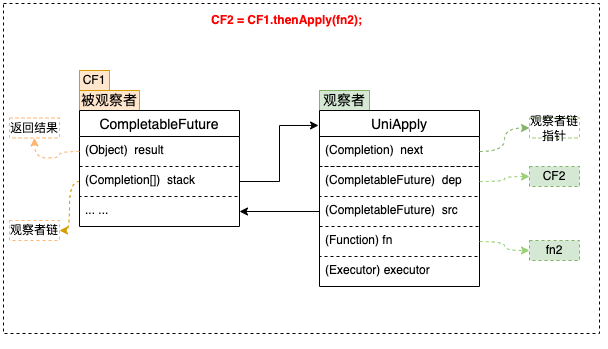
 如上代码块所示，父任务向executor线程池请求线程，并且内部子任务又向executor线程池请求线程。executor线程池大小为10，当同一时刻有10个请求到达，则executor线程池被打满，子任务请求线程时进入阻塞队列排队，但是父任务的完成又依赖于子任务，这时由于子任务得不到线程，父任务无法完成。主线程执行fatherFuture.join()进入阻塞状态，无法自行恢复。 故为解决该问题，需要将父任务与子任务做线程池隔离，两个任务请求不同的线程池，避免循环依赖导致的死锁。

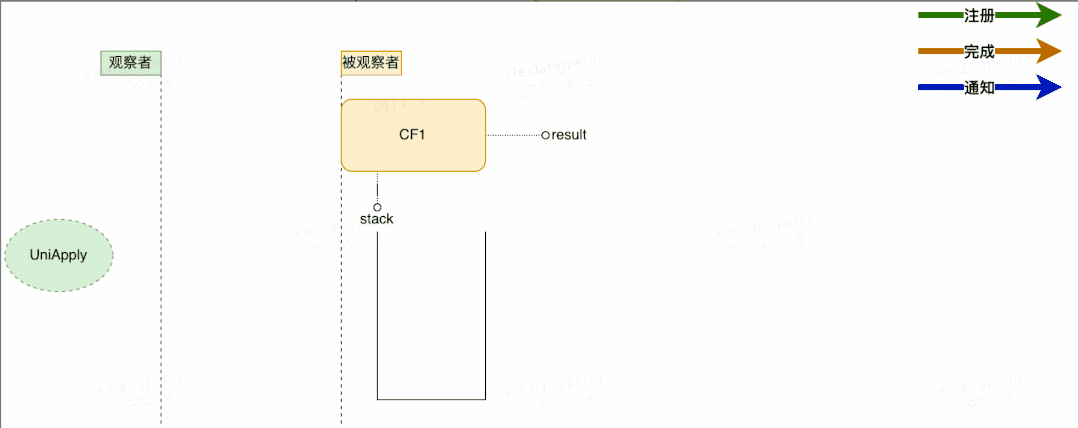
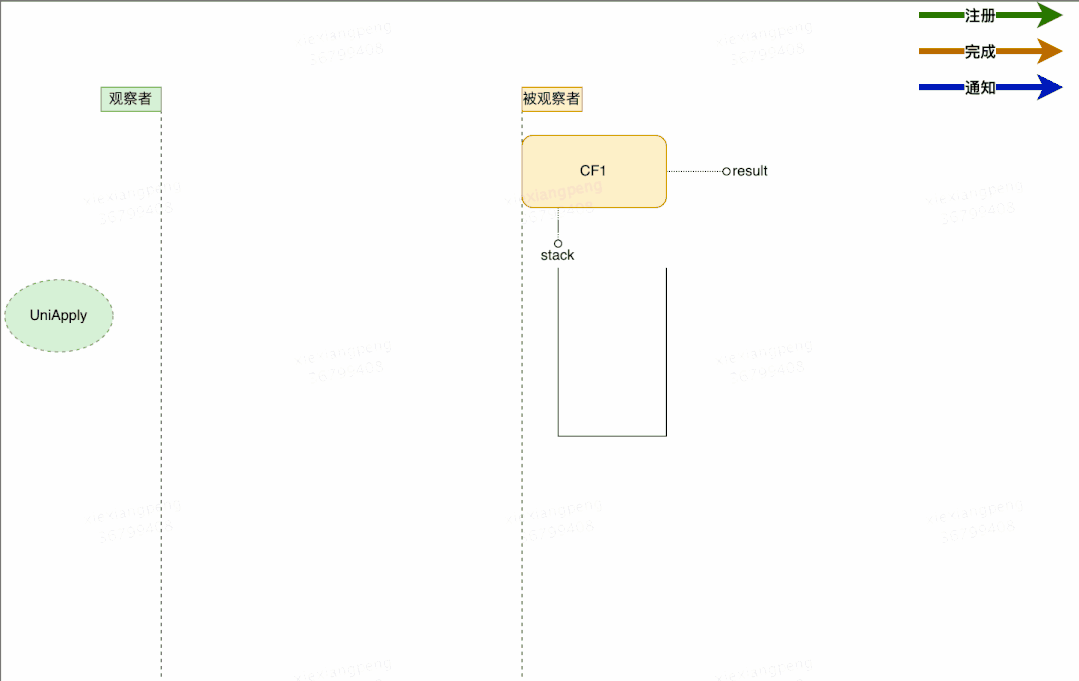
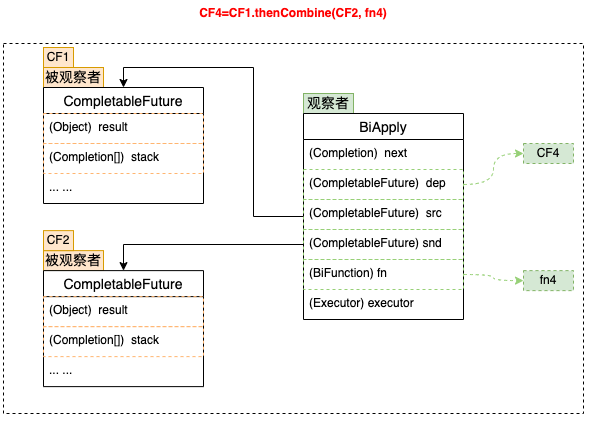
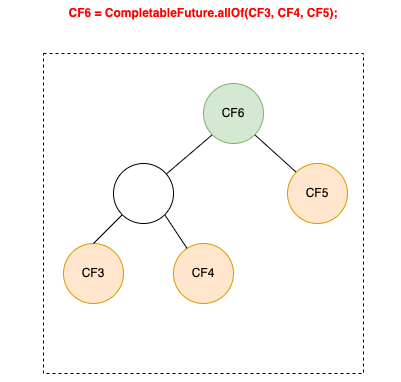
#### 3.3 多线程操作数据库的事务问题

由于事务是与线程绑定的，每一个线程与数据库建立链接时都会开启独立的事务，所以在多线程环境下操作数据库可能会存在事务不一致的问题。 解决思路:主线程开启多个子线程，每个子线程开启独立的事务,然后在数据库io后悬停线程,等待其他子线程全部执行完, 若任一一个子线程存在异常, 则全部子线程进行事务回滚.否则提交; 技术要点: 1.没有使用Object的wait和notify等方法进行线程间通信, 而是使用了LockSupport的park和unpark来实现线程间通信. 2.有兴趣的同学也可以尝试使用CountDownLatch（闭锁）来实现。

# 三.原理浅析

CompletableFuture中包含两个字段：**result**和**stack**。result用于存储当前CF的结果，stack（Completion）表示当前CF完成后需要触发的依赖动作（Dependency Actions），去触发依赖它的CF的计算，依赖动作可以有多个（表示有多个依赖它的CF），以栈（ [Treiber stack](https://en.wikipedia.org/wiki/Treiber_stack)）的形式存储，stack表示栈顶元素。  这种方式类似“观察者模式”，依赖动作（Dependency Action）都封装在一个单独Completion子类中。下面是Completion类关系结构图。CompletableFuture中的每个方法都对应了图中的一个Completion的子类，Completion本身是**观察者**的基类。

* UniCompletion继承了Completion，是一元依赖的基类，例如thenApply的实现类UniApply就继承自UniCompletion。
* BiCompletion继承了UniCompletion，是二元依赖的基类，同时也是多元依赖的基类。例如thenCombine的实现类BiRelay就继承自BiCompletion。  3.3.1 CompletableFuture的设计思想 按照类似“观察者模式”的设计思想，原理分析可以从“观察者”和“被观察者”两个方面着手。由于回调种类多，但结构差异不大，所以这里单以一元依赖中的thenApply为例，不再枚举全部回调类型。如下图所示：  **3.3.1.1 被观察者**

1. 每个CompletableFuture都可以被看作一个被观察者，其内部有一个Completion类型的链表成员变量stack，用来存储注册到其中的所有观察者。当被观察者执行完成后会弹栈stack属性，依次通知注册到其中的观察者。上面例子中步骤fn2就是作为观察者被封装在UniApply中。
2. 被观察者CF中的result属性，用来存储返回结果数据。这里可能是一次RPC调用的返回值，也可能是任意对象，在上面的例子中对应步骤fn1的执行结果。 **3.3.1.2 观察者** CompletableFuture支持很多回调方法，例如thenAccept、thenApply、exceptionally等，这些方法接收一个函数类型的参数f，生成一个Completion类型的对象（即观察者），并将入参函数f赋值给Completion的成员变量fn，然后检查当前CF是否已处于完成状态（即result != null），如果已完成直接触发fn，否则将观察者Completion加入到CF的观察者链stack中，再次尝试触发，如果被观察者未执行完则其执行完毕之后通知触发。
3. 观察者中的dep属性：指向其对应的CompletableFuture，在上面的例子中dep指向CF2。
4. 观察者中的src属性：指向其依赖的CompletableFuture，在上面的例子中src指向CF1。
5. 观察者Completion中的fn属性：用来存储具体的等待被回调的函数。这里需要注意的是不同的回调方法（thenAccept、thenApply、exceptionally等）接收的函数类型也不同，即fn的类型有很多种，在上面的例子中fn指向fn2。 3.3.2 整体流程 **3.3.2.1 一元依赖** 这里仍然以thenApply为例来说明一元依赖的流程：
6. 将观察者Completion注册到CF1，此时CF1将Completion压栈。
7. 当CF1的操作运行完成时，会将结果赋值给CF1中的result属性。
8. 依次弹栈，通知观察者尝试运行。  初步流程设计如上图所示，这里有几个关于注册与通知的并发问题，大家可以思考下： **Q1**：在观察者注册之前，如果CF已经执行完成，并且已经发出通知，那么这时观察者由于错过了通知是不是将永远不会被触发呢 ？ **A1**：不会。在注册时检查依赖的CF是否已经完成。如果未完成（即result == null）则将观察者入栈，如果已完成（result != null）则直接触发观察者操作。 **Q2**：在”入栈“前会有”result == null“的判断，这两个操作为非原子操作，CompletableFufure的实现也没有对两个操作进行加锁，完成时间在这两个操作之间，观察者仍然得不到通知，是不是仍然无法触发？ **A2**：不会。入栈之后再次检查CF是否完成，如果完成则触发。 **Q3**：当依赖多个CF时，观察者会被压入所有依赖的CF的栈中，每个CF完成的时候都会进行，那么会不会导致一个操作被多次执行呢 ？如下图所示，即当CF1、CF2同时完成时，如何避免CF3被多次触发。 **A3**：CompletableFuture的实现是这样解决该问题的：观察者在执行之前会先通过CAS操作设置一个状态位，将status由0改为1。如果观察者已经执行过了，那么CAS操作将会失败，取消执行。 通过对以上3个问题的分析可以看出，CompletableFuture在处理并行问题时，全程无加锁操作，极大地提高了程序的执行效率。我们将并行问题考虑纳入之后，可以得到完善的整体流程图如下所示：  CompletableFuture支持的回调方法十分丰富，但是正如上一章节的整体流程图所述，他们的整体流程是一致的。所有回调复用同一套流程架构，不同的回调监听通过**策略模式**实现差异化。 **3.3.2.2 二元依赖** 我们以thenCombine为例来说明二元依赖：  thenCombine操作表示依赖两个CompletableFuture。其观察者实现类为BiApply，如上图所示，BiApply通过src和snd两个属性关联被依赖的两个CF，fn属性的类型为BiFunction。与单个依赖不同的是，在依赖的CF未完成的情况下，thenCombine会尝试将BiApply压入这两个被依赖的CF的栈中，每个被依赖的CF完成时都会尝试触发观察者BiApply，BiApply会检查两个依赖是否都完成，如果完成则开始执行。这里为了解决重复触发的问题，同样用的是上一章节提到的CAS操作，执行时会先通过CAS设置状态位，避免重复触发。 **3.3.2.3 多元依赖** 依赖多个CompletableFuture的回调方法包括allOf、anyOf，区别在于allOf观察者实现类为BiRelay，需要所有被依赖的CF完成后才会执行回调；而anyOf观察者实现类为OrRelay，任意一个被依赖的CF完成后就会触发。二者的实现方式都是将多个被依赖的CF构建成一棵平衡二叉树，执行结果层层通知，直到根节点，触发回调监听。 

# 四.总结

1. CompletableFuture底层由于借助了魔法类Unsafe的相关CAS方法，除了get或join结果之外，其他方法都实现了无锁操作。
2. CompletableFuture实现了CompletionStage接口，因而具备了链式调用的能力，CompletionStage提供了either、apply、run以及then等相关方法，使得CompletableFuture可以使用各种应用场景。
3. CompletableFuture中有“源任务”和“依赖任务”，“源任务”的完成能够触发“依赖任务”的执行，这里的完成可以是返回正常结果或者是异常。
4. CompletableFuture默认使用ForkJoinPool，也可以使用指定线程池来执行任务。 事实上，如果每个操作都很简单的话（比如：上面的例子中按照id去查）没有必要用这种多线程异步的方式，因为创建线程还需要时间，还不如直接同步执行来得快。

事实证明，只有当每个操作很复杂需要花费相对很长的时间（比如，调用多个其它的系统的接口；比如，商品详情页面这种需要从多个系统中查数据显示的）的时候用CompletableFuture才合适，不然区别真的不大，还不如顺序同步执行。

上面就是CompletionStage接口中方法的使用实例，CompletableFuture同样也同样实现了Future，所以也同样可以使用get进行阻塞获取值，总的来说，CompletableFuture使用起来还是比较爽的，看起来也比较优雅一点。

## 五.参考文献

1.CompletableFuture (Java Platform SE 8 ) 2.CompletableFuture In Java With Examples 3.java - Does CompletionStage always wrap exceptions in CompletionException 4.CompletableFuture概述、创建方式、常用API、电商比价需求 5.多线程事务如何一起提交一起回滚(CountDownLatch实现)，并发处理批量数据，实现多线程事务回滚，事务补偿 6.CompletableFuture原理与实践-外卖商家端API的异步化