## CompletableFuture详解

### CompletableFuture 概念与例子

我们先把这两个单词拆开：CompletableFuture = Completable + Future，前面我们已经知道Future接口其实代表了一个异步的计算结果，通过它我们可以获取异步执行结果，或者等待异步执行结束，说白了就是把任务封装成Future然后丢到别的线程异步执行，保留一个存根，通过它可以拿到结果，而Completable是什么呢？翻译过来是可完成的。那么这里很明显了，就是一个可以异步执行的且可以明确完成的特殊future，那么它跟普通的Future有何区别呢？在继续聊它之前我们先看个例子。

/\*\*

\* 声明两个方法：

\* 1.getUserIdList：从数据库里获取用户id列表

\* 2.getUserGrade：调用远程REST API获取用户分数

\* 由于每个用户的信息都是独立获取的，所以这里可以并行执行。流程如下：

\* 1.将用户id列表转为stream

\* 2.通过map操作对每个用户id进行远程调用，将操作放入FJP中并行执行，返回

\* CompletableFuture

\* 3.将每个CompletableFuture通过allOf方法组合completableFutures生成一个新的CompletableFuture

\* 4.执行完毕后进行回调map方法获取每个用户的分数，封装为List

\* 5.封装完毕后，看看是否发生异常，如果没有异常那么输出用户的每个分数

\*/

static public void main(String[] args) throws Exception {

CompletableFuture[] completableFutures = getUserIdList().stream().map(uid -> getUserGrade(uid)).toArray(CompletableFuture[]：：new);

CompletableFuture.allOf(completableFutures).thenApply(v -> Stream.of(completableFutures).map(future -> {

try {

return future.get(); // 获取任务结果

} catch (Exception e) {

throw new RuntimeException(e);

}

}).collect(Collectors.toList())).whenComplete((userGradeList， e) -> {

if (e != null) {

throw new RuntimeException(e);

}

System.out.println(userGradeList);

}).join(); // 等待任务执行完毕

}

static List<String> getUserIdList() {

try {

Thread.sleep(500); // 模拟获取数据延迟

} catch (InterruptedException e) {

e.printStackTrace();

}

List<String> uidList = new ArrayList<>();

uidList.add("1");

uidList.add("2");

uidList.add("3");

return uidList;

}

static CompletableFuture<Integer> getUserGrade(String uid) {

return CompletableFuture.supplyAsync(() -> {

try {

Thread.sleep(500); // 模拟获取数据延迟

} catch (InterruptedException e) {

e.printStackTrace();

}

return 10;

});

// supplyAsync将任务放入线程池中执行，返回CompletableFuture

}

为了方便读者对比记忆，以上联合使用了流式编程和CompletableFuture的特性，读者可以仔细观察后发现，其实CompletableFuture和Stream思想是一样的：将多行变单行，触发回调执行。是的，就是这样，CompletableFuture就是将原来我需要自己编写的多线程处理转变为了流式编程。在任务执行完毕后触发回调，这个回调可以根据CompletableFuture中的xxxAsync或者xxx操作来选择是异步执行，还是同步执行，同时提供了allOf方法等待所有任务执行完毕再进行回调。如果说Stream把每一个操作叫做op（比如 ReduceOp），那么CompletableFuture就是将每一个操作叫做CompletionStage。

### CompletableFuture CompletionStage接口

我们先来看看CompletionStage接口。为了保证方法紧凑，这里我去掉了异步执行（xxxAsync）的操作，读者在了解了同步执行的方法后，在后面我们看到带有xxxAsync的方法，那么就是对于同步执行的异步操作，也即放入线程池中异步执行。CompletionStage接口描述如下。

public interface CompletionStage<T> {

// 当这个CompletionStage执行完毕后，返回一个新的CompletionStage，在这个新的CompletionStage中回调fn

public <U> CompletionStage<U> thenApply(Function<? super T，? extends U> fn);

// 当这个CompletionStage执行完毕后，返回一个新的CompletionStage，并在新的stage中调用action消费结果

public CompletionStage<Void> thenAccept(Consumer<? super T> action);

// 当这个CompletionStage执行完毕后，返回一个新的CompletionStage，并在新的stage中调用action

public CompletionStage<Void> thenRun(Runnable action);

// 当目前的CompletionStage和thenCombine传入的thenCombine正常完成后，返回一个新的CompletionStage，并在新的stage中调用fn函数

public <U，V> CompletionStage<V> thenCombine (CompletionStage<? extends U> other， BiFunction<? super T，? super U，? extends V> fn);

// 当目前的CompletionStage和thenCombine传入的other正常完成后，返回一个新的CompletionStage，并在新的stage中调用action消费结果，注意这里的没有返回值

public <U> CompletionStage<Void> thenAcceptBoth

(CompletionStage<? extends U> other，

BiConsumer<? super T， ? super U> action);

// 当目前的CompletionStage和other传入的thenCombine正常完成后，返回一个新的CompletionStage，并执行action

public CompletionStage<Void> runAfterBoth(CompletionStage<?> other，Runnable action);’

// 当目前的CompletionStage和传入的CompletionStage其中一个正常完成后，返回一个新的CompletionStage，并执行fn

public <U> CompletionStage<U> applyToEither (CompletionStage<? extends T> other，Function<? super T， U> fn);

// 当目前的CompletionStage和传入的CompletionStage其中一个正常完成后，返回一个新的CompletionStage，并执行action

public CompletionStage<Void> acceptEither (CompletionStage<? extends T> other， Consumer<? super T> action);

// 当目前的CompletionStage和传入的CompletionStage其中一个正常完成后，返回一个新的CompletionStage，并执行action

public CompletionStage<Void> runAfterEither(CompletionStage<?> other，Runnable action);

// 当目前的CompletionStage正常完成，然后返回一个新的CompletionStage，并且在其中调用fn，注意这里和thenApply的区别在于：thenApply中fn返回的类型没有限制而这里的thenCompose限制返回类型必须为CompletionStage的子类

public <U> CompletionStage<U> thenCompose (Function<? super T， ? extends CompletionStage<U>> fn);

// 当目前的CompletionStage执行发生异常时，返回一个新的CompletionStage，并且执行其中的fn函数

public CompletionStage<T> exceptionally (Function<Throwable， ? extends T> fn);

// 当目前的CompletionStage执行完成时，无论是否发生异常均返回一个新的CompletionStage，并且执行其中的action消费函数

public CompletionStage<T> whenComplete (BiConsumer<? super T， ? super Throwable> action);

// 当目前的CompletionStage执行完成时，无论是否发生异常均返回一个新的CompletionStage，并且执行其中的fn函数

public <U> CompletionStage<U> handle (BiFunction<? super T， Throwable， ? extends U> fn);

// 将当前CompletionStage转变为CompletableFuture

public CompletableFuture<T> toCompletableFuture();

}

观察一下规律，所有的操作都封装成了一个CompletionStage，一个CompletionStage可以和另外一个CompletionStage产生关系，并且生成新的CompletionStage，并且可以设置发生异常或者正常完成时的操作，注意，这里操作分为三类：Function（有入参和出参）、Consumer（只有入参）、Runnable（没有入参和出参，仅仅执行动作）。并且我在其中省略了Async的异步操作，代表不同的CompletionStage可以在不同的线程池中执行。

### CompletableFuture runAsync原理

现在，我们可以来看看CompletableFuture的实现原理了。我们还是得以一个简单的例子开始。

CompletableFuture.runAsync(() -> System.out.println("hello")).join(); // 异步执行放入的Runnable对象

很简单，异步执行方法，调用join等待执行完毕。来看runAsync源码实现。

public static CompletableFuture<Void> runAsync(Runnable runnable) {

/\*\*

\* private static final Executor asyncPool = useCommonPool ?

\* ForkJoinPool.commonPool() ： new ThreadPerTaskExecutor();

\* useCommonPool默认为true，所以放到FJP中执行

\*/

return asyncRunStage(asyncPool， runnable);

}

static CompletableFuture<Void> asyncRunStage(Executor e， Runnable f) {

if (f == null) throw new NullPointerException();

// 封装一个返回值为空的CF

CompletableFuture<Void> d = new CompletableFuture<Void>();

// 将d和f封装为AsyncRun放入线程池执行，这里想想为何要把d也封装进去呢？很明显，因为我们可能需要在执行完毕后触发某个动作对吧

e.execute(new AsyncRun(d， f));

// 返回刚刚生成的d

return d;

}

// 继承自我们亲密无间，无比熟悉的FJT

static final class AsyncRun extends ForkJoinTask<Void> implements Runnable，AsynchronousCompletionTask {

// 当前任务依赖的CF

CompletableFuture<Void> dep;

Runnable fn; // 执行函数

AsyncRun(CompletableFuture<Void> dep， Runnable fn) {

this.dep = dep; this.fn = fn;

}

public final Void getRawResult() { return null; }

public final void setRawResult(Void v) {}

public final boolean exec() { run(); return true; }

public void run() {

CompletableFuture<Void> d; Runnable f;

// 开始执行咱们的fn

if ((d = dep) != null && (f = fn) != null) {

dep = null; fn = null;

// 当d还没有完成的时候才开始执行，避免重复执行

if (d.result == null) {

try {

f.run();

/\*\*

\* final boolean completeNull() {

\* return Unsafe.compareAndSwapObject(this，

\* RESULT， null，NIL);

\* }

\* 写上一个NIL对象：AltResult NIL = new

\* AltResult(null);null表明没有出现异常

\*/

d.completeNull();

} catch (Throwable ex) {

d.completeThrowable(ex);

}

}

// 调用postComplete完成CF

d.postComplete();

}

}

}

我们看到当我们调用runAsync异步执行时，是将任务Runnable对象封装成AsyncRun对象放入线程池中执行。而AsyncRun继承自FJT，所以兼容在FJP中执行，当然AsyncRun也实现了Runnable接口，意味着我们也可以将其放到我们自定义线程池中执行。

### CompletableFuture postComplete原理

我们看到当AsyncRun在线程池执行完成后，将会调用依赖的CompletableFuture 的postComplete方法完成回调操作。接下来我们来看当异步任务完成后回调依赖的CF完成链式操作的执行原理。

final void postComplete() {

CompletableFuture<?> f = this; Completion h;

/\*\*

\* 当前f所指CF中的stack不为空 或者 f变量和当前CF不匹配且当前任务的stack不

\* 为空时执行。为什么会出现这样的情况？

\* 来看一个场景：A->B->C;A->D

\* 1.A执行完毕后执行B，B执行完毕后执行C。A执行完毕后也执行D

\* 那么由于我们是调用栈的关系，所以这时肯定是先执行D的Completion

\* 2.这时这里的变量f为A，h为D的Completion，接下来执行D的Completion且栈

\* 顶变为D.next为B的Completion

\* 3.再一次循环，这时f为A，h为B的Completion，执行B的Completion，这时由

\* 于B的Completion有依赖C的CF变量，这时tryFire返回了C不为null，那么返回

\* C作为变量f

\* 4.再一次循环，这时f为C，h为C的stack的Completion，此时f != this，那

\* 么将C的Completion压入到A的栈顶

\* 5.继续循环，此时C的stack为null，且f（此时为C）!= this 那么将f变为A，

\* A的stack为C的Completion，这时h为C的Completion，执行C的Completion

\* 6.结束循环

\*/

while ((h = f.stack) != null ||

(f != this && (h = (f = this).stack) != null)) {

CompletableFuture<?> d; Completion t;

// 取出栈顶的任务，并且将栈指针stack指向下一个任务

if (f.casStack(h， t = h.next)) {

// 取出的Completion t不为空

if (t != null) {

// 如果f变量和当前CF不是同一个，那么将h压回到栈顶

if (f != this) {

pushStack(h);

continue;

}

// h.next Completion已经取出来放到f所指的CF的stack中，这里不需要再保存next Completion

h.next = null;

}

// 调用上面取出的Completion的tryFire方法，如果返回结果为空，那么返回当前CF，否则返回d

f = (d = h.tryFire(NESTED)) == null ? this ： d;

}

}

}

可以看到执行完毕后回调Completion执行回调动作，那么什么是Completion呢。

### CompletableFuture Completion原理

Completion英文翻译意思：完成。那么完成什么呢？来看，这里继承自FTJ，那么表明这个完成代表了一个动作，且这个动作可以放到FJP线程池中执行，同时，这里包含了一个next变量，同样也是一个Completion，将一串完成动作串起来了，同时这个结构是通过链表实现的栈结构。实现原理如下。

abstract static class Completion extends ForkJoinTask<Void>

implements Runnable， AsynchronousCompletionTask {

volatile Completion next;

/\*\*

\* 如果触发，则执行完成操作，返回可能需要传播的依赖项

\* 参数可选：

\* 1.SYNC = 1 同步执行

\* 2.ASYNC = 0 异步执行

\* 3.NESTED = -1 嵌套执行

\*/

abstract CompletableFuture<?> tryFire(int mode);

// 如果依然还有动作需要执行，那么返回true

abstract boolean isLive();

// 执行tryFire，注意这里参数为ASYNC

public final void run() { tryFire(ASYNC); }

public final boolean exec() { tryFire(ASYNC); return true; }

public final Void getRawResult() { return null; }

public final void setRawResult(Void v) {}

}

由于上面的例子并没有添加回调动作，那么这里给出一个带有回调动作的例子。

CompletableFuture.runAsync(()->System.out.println("hello")).thenRun(()->System.out.println("over")).join(); // 执行完成后输出over操作

在上面我们看到asyncRunStage方法中返回了一个CompletableFuture<Void>代表无返回值的CF，那么我们这个的thenRun就是针对这个CF执行的，来看源码。

public CompletableFuture<Void> thenRun(Runnable action) {

// 注意这里传入的executor为null，表明同步执行

return uniRunStage(null， action);

}

private CompletableFuture<Void> uniRunStage(Executor e， Runnable f) {

if (f == null) throw new NullPointerException();

// 生成一个新的CF

CompletableFuture<Void> d = new CompletableFuture<Void>();

/\*\*

\* 执行线程池不为空或者调用uniRun执行失败，那么封装一个UniRun对象，调用push压入

\* 当前CF的stack中。读者可能不太理解这里的为什么把新生成的CF称之为d，这里我给出d

\* 的全称：dependent，这下明了了吧，叫做依赖项，确实是这样，毕竟需要等待当前CF执

\* 行完毕才能执行。而且注意，这里如果传入了Executor，那么表明一定要异步执行，这时

\* 必须封装UniRun，因为线程池中执行的必须是一个任务而不是一个CF。（在我们的例子中

\* 仅仅只是输出了字符串，所以可能进入这里直接就d.uniRun执行输出over的操作）

\*/

if (e != null || !d.uniRun(this， f， null)) {

UniRun<T> c = new UniRun<T>(e， d， this， f);

push(c); // 压入依赖栈

c.tryFire(SYNC); // 尝试执行

}

return d;

}

// 执行传入的Runnable

final boolean uniRun(CompletableFuture<?> a， Runnable f， UniRun<?> c) {

Object r; Throwable x;

// 父任务为空、父任务还没有执行完成、函数f为空均不执行，如果这时父CF a执行完了会立即调用f

if (a == null || (r = a.result) == null || f == null)

return false;

// 结果当前CF结果为null，表明没有执行完成

if (result == null) {

if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)

// 如果执行成功且包含异常，那么将异常赋值给当前CF的结果

else

completeThrowable(x， r);

try {

// 如果任务UniRun c（completion）不为空，那么调用claim确认下是否应该执行该completion。如果不能执行，那么直接返回false

if (c != null && !c.claim())

return false;

f.run(); // 否则调用函数f

completeNull(); // 无返回值完成

} catch (Throwable ex) {

completeThrowable(ex);

}

}

return true;

}

// 将Completion c压入当前CF的stack调用栈中

final void push(UniCompletion<?，?> c) {

if (c != null) {

// 当前CF还未完成且tryPushStack调用失败

while (result == null && !tryPushStack(c))

/\*\*

\* static void lazySetNext(Completion c， Completion next) {

\* Unsafe.putOrderedObject(c， NEXT， next);

\* }

\*/

// 如果CAS失败，将C的next恢复为null

lazySetNext(c， null);

}

}

// 尝试通过CAS替换栈顶，读者可能会问：为什么是栈？想想LIFO（先进后出），不就正好满足咱们的要求吗？最后放入的action先执行

final boolean tryPushStack(Completion c) {

Completion h = stack; // 当前栈顶引用

lazySetNext(c， h); // 设置c的next为当前栈顶

return Unsafe.compareAndSwapObject(this， STACK， h， c); // CAS替换栈顶引用

}

好的，这下很清晰了：生成了一个新的CF，然后通过传入的executor是不是null，看看是不是异步执行，如果不是，那么尝试调用uniRun直接执行回调动作，当然，如果此时父CF还未执行完成，那么将会通过push方法将任务放入栈顶，父CF执行完毕后弹出执行。注意，这里有一个临界情况，如果在push方法的while循环中任务执行成功了，那么这时任务没被压入栈中，那么谁来执行呢？很明显。uniRunStage方法中push方法后边调用了一次c.tryFire(SYNC)来解决这种临界条件。

现在问题很明了了，集中在Completion的子类UniRun类中。这里直接看源码。

// 拥有一个 src：源CF 和 dep：关联的CF 和 executor：异步执行器

abstract static class UniCompletion<T，V> extends Completion {

Executor executor; // 异步执行时使用的执行器

CompletableFuture<V> dep; // 关联的CF

CompletableFuture<T> src; // 父CF

UniCompletion(Executor executor， CompletableFuture<V> dep，CompletableFuture<T> src) {

this.executor = executor; this.dep = dep; this.src = src;

}

// 判断线程应该执行

final boolean claim() {

Executor e = executor;

// 使用FJT的tag标记位来实现只有一个线程可以设置执行

if (compareAndSetForkJoinTaskTag((short)0， (short)1)) {

// 如果执行器为空，那么直接返回true

if (e == null)

return true;

// 任务执行前将executor的引用释放，这时只需要局部变量的e持有引用

executor = null;

// 调用线程池执行当前Completion

e.execute(this);

}

return false;

}

// 是否有回调动作，如果关联的CF不为空的话，可能还有需要执行的Completion

final boolean isLive() { return dep != null; }

}

static final class UniRun<T> extends UniCompletion<T，Void> {

Runnable fn;

UniRun(Executor executor， CompletableFuture<Void> dep，

CompletableFuture<T> src， Runnable fn) {

// 初始化父类UniCompletion

super(executor， dep， src); this.fn = fn;

}

// 开始执行当前Completion

final CompletableFuture<Void> tryFire(int mode) {

CompletableFuture<Void> d; CompletableFuture<T> a;

// 如果关联的CF为空那么直接返回null，否则尝试执行依赖项的uniRun方法，如果执行失败同样返回

// 只有ASYNC异步执行才会将传入的UniRun设置为空

if ((d = dep) == null ||

!d.uniRun(a = src， fn， mode > 0 ? null ： this))

return null;

// 释放引用

dep = null; src = null; fn = null;

// 成功执行后，将回调关联CF，让它执行调用栈stack的任务

return d.postFire(a， mode);

}

}

// CompletableFuture方法，a为父CF，mode为触发执行Completion时的模式

final CompletableFuture<T> postFire(CompletableFuture<?> a， int mode) {

// 父CF不为空且执行栈不为空

if (a != null && a.stack != null) {

// NEST嵌套执行或者父CF还未完成

if (mode < 0 || a.result == null)

// 遍历调用栈并清除已经完成的Completion

a.cleanStack();

else

// 执行父任务的调用栈的Completion

a.postComplete();

}

// 当前任务已经执行完毕且调用栈不为空

if (result != null && stack != null) {

// 嵌套执行，返回当前CF对象，让根任务来执行当前CF的执行栈

if (mode < 0)

return this;

else

// 其它模式执行当前CF的调用栈

postComplete();

}

return null;

}

看完了这个流程还是得以一个例子来说明这里面发生了什么，方便读者进行理解。

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

System.out.println("hello");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

future.thenRun(() -> System.out.println("over1")).thenRun(() -> System.out.println("over11"));

future.thenRun(() -> System.out.println("over2")).thenRun(() -> System.out.println("over22")).join();

运行结果：

hello

over2

over22

over1

over11

工作流程如下所示。

1. 主线程runAsync方法中生成了一个新的CF-1，然后创建了一个FTJ任务AsyncRun，提交到FJP中执行
2. 在FJP中执行AsyncRun的run方法，进而执行fn，此时当前线程睡眠1000
3. 主线程继续执行，此时调用CF-1的thenRun方法，创建了一个新的CF-2，因为此时CF-1还没有执行完，那么创建一个Completion UniRun对象，将它放到CF-1的stack变量中
4. 主线程继续执行，调用CF-2的thenRun方法，创建了一个新的CF-3，因为此时CF-2还没有执行完，那么创建一个Completion UniRun对象，将它放到CF-2的stack变量中。此时状态：CF-1.stack -> UniRun ( CF-2.stack -> UniRun( CF-3.stack -> null ) ) ; CF-1.stack = null
5. 主线程继续执行，此时调用CF-1的thenRun方法，创建了一个新的CF-4，因为此时CF-1还没有执行完，那么创建一个Completion UniRun对象，将它放到CF-1的stack变量中
6. 主线程继续执行，调用CF-4的thenRun方法，创建了一个新的CF-5，因为此时CF-4还没有执行完，那么创建一个Completion UniRun对象，将它放到CF-4的stack变量中。此时状态：CF-1.stack -> UniRun ( CF-4.stack -> UniRun( CF-5.stack -> null ) ) ；CF-1.stack.next -> UniRun ( CF-2.stack -> UniRun( CF-3.stack -> null ) )
7. 此时FJP中的线程执行完毕AsyncRun传入的匿名函数，输出"hello"，将CF-1的结果设置为AltResult对象
8. 调用CF-1的postComplete方法，发现stack不为空，那么这时取出stack栈顶的Completion： UniRun ( CF-4.stack -> UniRun( CF-5.stack -> null ) ) 并设置stack变量为UniRun ( CF-2.stack -> UniRun( CF-3.stack -> null ) )
9. 开始执行UniRun的tryFire方法，mode为NEST嵌套执行，然后执行传递进入的fn，输出"over2"，将CF-4的结果设置为AltResult对象
10. 执行完毕后调用CF-4的postFire方法，此时CF-4的结果为AltResult非空且mode为NEST，这时返回CF-4的引用给CF-1的postComplete方法循环局部变量f中
11. 继续执行CF-1的postComplete方法，这时f为CF-4且它的stack不为空，指向UniRun( CF-5.stack -> null )，由于UniRun的next为null，这时继续执行UniRun，这时输出"over22"，将CF-5的结果设置为AltResult对象，由于CF-5的stack为null，这时返回null，那么修正f变量指向CF-1
12. 循环以上过程进而输出over1，over11

是时候该小结一下了，不然是不是感觉脑子嗡嗡的。每个CompletableFuture对象都是一个CompletableStage且包含了两个核心变量：result（执行结果）、stack（执行完毕后回调的动作Completion。由于CF不是任务，这时需要一个任务来包装执行，这就是上面出现的AsyncRun。所以这里的FJT的子类包含两种类型：1.Completion 2.AsyncRun。当提交到FJP的任务AsyncRun执行完毕后，设置所包含的CF也即CompletableStage为完成状态，然后触发CF所包含的Completion即可。

那么上面我们是同步执行的例子，这里再给出一个异步执行的例子。

// 注意这里执行完调用的是thenRunAsync而不是thenRun

CompletableFuture.runAsync(()->System.out.println("hello")).thenRunAsync(()->System.out.println("over")).join();

上面通过源码我们看到了：thenRun中的fn执行是在FJP的线程中执行和输出"hello"的线程是同一个（通过调用CF的postComplete，然后调用UniRun的tryFire执行），而这里的thenRunAsync会将后面输出“over“的fn放到另外的线程中异步执行。来先看源码。

public CompletableFuture<Void> thenRunAsync(Runnable action) {

// 注意这里传入了asyncPool FJP线程池

return uniRunStage(asyncPool， action);

}

private CompletableFuture<Void> uniRunStage(Executor e， Runnable f) {

if (f == null) throw new NullPointerException();

CompletableFuture<Void> d = new CompletableFuture<Void>();

// 和thenRun的区别在于，这里由于e不为null，所以直接生成UniRun

if (e != null || !d.uniRun(this， f， null)) {

UniRun<T> c = new UniRun<T>(e， d， this， f);

push(c);

c.tryFire(SYNC);

}

return d;

}

// 在AsyncRun中最后调用了d.postComplete()，进而调用UniRun的tryFire(NESTED)方法：

final CompletableFuture<Void> tryFire(int mode) {

CompletableFuture<Void> d; CompletableFuture<T> a;

// 注意这里的mode为NESTED=-1，所以传入UniRun对象，线程池存在将会提交执行，并且返回false，所以该方法返回null

if ((d = dep) == null || !d.uniRun(a = src， fn， mode > 0 ? null ： this))

return null;

dep = null; src = null; fn = null;

return d.postFire(a， mode);

}

// CF中的方法，前面也看到了用于执行UniRun 这个 Completion，特别注意这里的c不为空

final boolean uniRun(CompletableFuture<?> a， Runnable f， UniRun<?> c) {

Object r; Throwable x;

if (a == null || (r = a.result) == null || f == null)

return false;

if (result == null) {

if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)

completeThrowable(x， r);

else

try {

// 这里是重点，如果线程池存在那么提交执行，那么返回false

if (c != null && !c.claim())

return false;

f.run();

completeNull();

} catch (Throwable ex) {

completeThrowable(ex);

}

}

return true;

}

// UniRun的方法

final boolean claim() {

Executor e = executor;

if (compareAndSetForkJoinTaskTag((short)0， (short)1)) {

if (e == null)

return true;

executor = null;

// 线程池中执行

e.execute(this);

}

return false; // 返回false

}

接下来，可以从源码角度来理解咱们的例子中的CF的使用方式了。我们逐步来分解。我们先来看getUserGrade方法。

static CompletableFuture<Integer> getUserGrade(String uid) {

return CompletableFuture.supplyAsync(() -> {

try {

// 模拟获取数据延迟

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

return 10;

});

// supplyAsync将任务放入线程池中执行，返回CompletableFuture

}

// 和thenRunAsync唯一不同的是这里的参数为一个Supplier而不是一个Runnable

public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {

return asyncSupplyStage(asyncPool， supplier);

}

static <U> CompletableFuture<U> asyncSupplyStage(Executor e，Supplier<U> f) {

if (f == null) throw new NullPointerException();

// 每一步都是一个CompletableStage，所以新生成一个CF

CompletableFuture<U> d = new CompletableFuture<U>();

// 这里不再是AsyncRun，变为了AsyncSupply代表f有返回值的异步任务

e.execute(new AsyncSupply<U>(d， f));

return d;

}

static final class AsyncSupply<T> extends ForkJoinTask<Void>

implements Runnable， AsynchronousCompletionTask {

CompletableFuture<T> dep; Supplier<T> fn;

AsyncSupply(CompletableFuture<T> dep， Supplier<T> fn) {

this.dep = dep; this.fn = fn;

}

public final Void getRawResult() { return null; }

public final void setRawResult(Void v) {}

public final boolean exec() { run(); return true; }

public void run() {

CompletableFuture<T> d; Supplier<T> f;

if ((d = dep) != null && (f = fn) != null) {

dep = null; fn = null;

if (d.result == null) {

try {

// 调用Supplier获取结果，并将结果放到当前绑定的CF中。对应到我们例子中就是用户的分数

d.completeValue(f.get());

} catch (Throwable ex) {

d.completeThrowable(ex);

}

}

// 同样执行依赖的CF的完成回调

d.postComplete();

}

}

}

### CompletableFuture allOf原理

接下来是CompletableFuture.allOf(completableFutures)。该方法可以将所有的completableFutures组合成一个新的CompletableFuture，我们可以通过该方法返回的CompletableFuture等待所有任务执行完成。

public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) {

return andTree(cfs， 0， cfs.length - 1);

}

static CompletableFuture<Void> andTree(CompletableFuture<?>[] cfs，int lo，int hi) {

// 构建一个新的stage

CompletableFuture<Void> d = new CompletableFuture<Void>();

// 如果lo大于hi，表明数组为空，那么直接将结果设置为AltResult

if (lo > hi)

d.result = NIL;

else {

CompletableFuture<?> a， b;

// 找到中间索引，并分割左右CF

int mid = (lo + hi) >>> 1;

if ((a = (lo == mid ? cfs[lo] ：andTree(cfs， lo， mid))) == null ||

(b = (lo == hi ? a ： (hi == mid+1) ? cfs[hi] ：andTree(cfs， mid+1， hi))) == null)

throw new NullPointerException();

// 判断a和b是否执行完毕

if (!d.biRelay(a， b)) {

// 创建BiRelay Completion

BiRelay<?，?> c = new BiRelay<>(d， a， b);

// 将BiRelay Completion放入到a任务的stack回调栈中

a.bipush(b， c);

// 为了避免添加进去期间任务已经执行完成，尝试手动执行一次，mode为SYNC同步执行

c.tryFire(SYNC);

}

}

return d;

}

接下来我们来看biRelay方法如何查询a和b CompletableFuture是否完成。实现过程如下。

boolean biRelay(CompletableFuture<?> a，CompletableFuture<?> b) {

Object r， s; Throwable x;

// a或者b为null直接返回false，a和b任务和一个没有执行完毕返回false

if (a == null || (r = a.result) == null ||

b == null || (s = b.result) == null)

return false;

// 当前CF还没有执行

if (result == null) {

// 检测a和b执行的结果是否包含异常，如果含有异常，那么将异常结果设置为当前CF的结果

if (r instanceof AltResult && (x = ((AltResult)r).ex) != null)

completeThrowable(x， r);

else if (s instanceof AltResult && (x = ((AltResult)s).ex) != null)

completeThrowable(x， s);

else

// 正常结束

completeNull();

}

return true;

}

当我们发现a和b还未完成时，调用bipush方法将BiRelay Completion放入到a任务的stack回调栈中。实现过程如下。

final void bipush(CompletableFuture<?> b， BiCompletion<?，?，?> c) {

if (c != null) {

Object r;

// 将c Completion压入当前stack中

while ((r = result) == null && !tryPushStack(c))

lazySetNext(c， null);

// b CF还没有完成

if (b != null && b != this && b.result == null) {

// 如果当前CF还未完成，那么封装CoCompletion赋值给q，否则直接将c赋值给q，将q压入b的stack中

Completion q = (r != null) ? c ： new CoCompletion(c);

while (b.result == null && !b.tryPushStack(q))

lazySetNext(q， null);

}

}

}

接下来我们来看上面用到的两个Completion：BiCompletion、BiRelay的定义。

// 包含两个src CompletableFuture的特殊UniCompletion

abstract static class BiCompletion<T，U，V> extends UniCompletion<T，V> {

CompletableFuture<U> snd;

BiCompletion(Executor executor， CompletableFuture<V> dep，

CompletableFuture<T> src， CompletableFuture<U> snd) {

super(executor， dep， src); this.snd = snd;

}

}

// 用于组合两个源CompletableFuture的结果，并且回调关联的CompletableFuture

static final class BiRelay<T，U> extends BiCompletion<T，U，Void> {

BiRelay(CompletableFuture<Void> dep，CompletableFuture<T> src，CompletableFuture<U> snd) {

// 强制设置executor为null

super(null， dep， src， snd);

}

final CompletableFuture<Void> tryFire(int mode) {

CompletableFuture<Void> d;

CompletableFuture<T> a;

CompletableFuture<U> b;

// 回调依赖的CF的biRelay方法完成CF

if ((d = dep) == null || !d.biRelay(a = src， b = snd))

return null;

src = null; snd = null; dep = null;

// 执行成功后回调

return d.postFire(a， b， mode);

}

}

这里可见将传入的所有CompletableFuture通过BiRelay Completion组合成一个回调执行树。拿我们这里的例子来说，CompletableFuture有三个。

// getUserIdList方法返回三个uid。从而生成三个CompletableFuture

List<String> uidList = new ArrayList<>();

uidList.add("1");

uidList.add("2");

uidList.add("3");

那么同样来描述一下流程。

1. 主线程调用allOf方法，间接调用andTree，这时cfs为上面三个uid调用getUserGrade生成的CompletableFuture数组，lo为0，hi为2
2. 创建一个新的CompletableFuture<Void> d1，计算mid中间值 mid = (0 + 2) >>> 1 -> 1
3. 此时 lo != mid，那么递归一次进入andTree，这时lo为0，hi为1
4. 创建一个新的CompletableFuture<Void> d2，计算mid中间值 mid = (0 + 1) >>> 1 -> 0
5. 此时 lo == mid 将变量a = cfs[0]，b = cfs[1]
6. 假如此时a和b都没有完成，那么d2.biRelay(a， b)将返回false，那么创建BiRelay2( d2，a = cfs[0]，b = cfs[1] ) Completion，压入a的stack中，返回d2
7. 此时在第一次进入andTree的 a = d2，b = cfs[3]
8. 继续调用d.biRelay(a， b)，假如此时 a = d2 或者 b = cfs[3] 都没有完成
9. 那么继续创建一个新的BiRelay1(d1，a = d2，b = cfs[3]) Completion，将其压入 a = d2 的stack中返回d1
10. 如果此时，cfs[0]完成执行了，那么回调BiRelay2 Completion，这时将执行BiRelay2的tryFire方法，间接调用d2的biRelay，因为此时cfs[1]未完成，所以直接返回
11. 如果此时，cfs[1]完成执行了，那么回调BiRelay2 Completion，这时将执行BiRelay2的tryFire方法，间接调用d2的biRelay，因为此时cfs[0]已经完成，所以设置d2完成
12. 由于d2完成，那么回调BiRelay1 Completion，由于cfs[3]还未完成，这时直接返回
13. 如果此时，cfs[3]完成执行了，那么回调BiRelay1 Completion，由于d2已经完成，那么设置d1也为完成状态，然后回调d1的postFire方法继续执行设置的Completion

接下来我们继续看thenApply方法做了什么。

CompletableFuture.allOf(completableFutures)

.thenApply(v -> Stream.of(completableFutures).map(future -> {

try {

return future.get(); // 获取任务结果

} catch (Exception e) {

throw new RuntimeException(e);

}

}).collect(Collectors.toList())).whenComplete((userGradeList， e) -> {

if (e != null) {

throw new RuntimeException(e);

}

System.out.println(userGradeList);

}).join(); // 等待任务执行完毕

// CF的thenApply方法

public <U> CompletableFuture<U> thenApply(

// 注意这里是Function有输入和输出

Function<? super T，? extends U> fn) {

// 传入Executor为null，表明同步执行（FJP中执行TASK回调tryFire方法的线程）

return uniApplyStage(null， fn);

}

private <V> CompletableFuture<V> uniApplyStage(

Executor e， Function<? super T，? extends V> f) {

if (f == null) throw new NullPointerException();

// 生成新的stage

CompletableFuture<V> d = new CompletableFuture<V>();

// 此时d还未完成不调用

if (e != null || !d.uniApply(this，f，null)) {

// 生成UniApply Completion压入当前CF的stack中

UniApply<T，V> c = new UniApply<T，V>(e，d，this，f);

push(c);

c.tryFire(SYNC);

}

return d;

}

// 包含一个fn的特殊UniCompletion

static final class UniApply<T，V> extends UniCompletion<T，V> {

Function<? super T，? extends V> fn;

UniApply(Executor executor， CompletableFuture<V> dep，CompletableFuture<T> src， Function<? super T，? extends V> fn) {

super(executor， dep， src); this.fn = fn;

}

// 当CF完成后回调

final CompletableFuture<V> tryFire(int mode) {

CompletableFuture<V> d; CompletableFuture<T> a;

// 调用uniApply

if ((d = dep) == null ||

!d.uniApply(a = src， fn， mode > 0 ? null ： this))

return null;

dep = null; src = null; fn = null;

return d.postFire(a， mode);

}

}

// 执行传入的函数f

final <S> boolean uniApply(CompletableFuture<S> a，Function<? super S，?extends T> f， UniApply<S，T> c) {

Object r; Throwable x;

if (a == null || (r = a.result) == null || f == null)

return false; // 所压入的CF必须执行完成才能回调

tryComplete： if (result == null) { // 当前关联的cf必须还未执行完成

// 如果压入的CF结果包含异常，那么设置异常结果后返回。注意这里用java的标签当C语言的goto使用

if (r instanceof AltResult) {

if ((x = ((AltResult)r).ex) != null) {

completeThrowable(x， r);

break tryComplete;

}

// 否则不持有a的结果引用，帮助GC执行

r = null;

}

try {

// 是异步执行，那么这里通过claim提交线程池执行

if (c != null && !c.claim())

return false;

// 将r转为函数接受的泛型s，注意这里为null

@SuppressWarnings("unchecked") S s = (S) r;

// 然后调用函数设置当前cf的完成结果为函数返回值

completeValue(f.apply(s));

} catch (Throwable ex) {

completeThrowable(ex);

}

}

return true;

}

接下来是WhenComplete方法的执行过程。

CompletableFuture.allOf(completableFutures)

.thenApply(v -> Stream.of(completableFutures).map(future -> {

try {

// 获取任务结果

return future.get();

} catch (Exception e) {

throw new RuntimeException(e);

}

}).collect(Collectors.toList()))

// 当CF执行完后回调，userGradeList为CF的运行结果，e为执行过程中的异常

.whenComplete((userGradeList， e) -> {

if (e != null) {

throw new RuntimeException(e);

}

System.out.println(userGradeList);

// 等待任务执行完毕

}).join();

// CF方法，执行完回调

public CompletableFuture<T> whenComplete(

// 注意这里为BiConsumer表明两个输入项：结果、异常

BiConsumer<? super T， ? super Throwable> action) {

// 同步执行

return uniWhenCompleteStage(null， action);

}

private CompletableFuture<T> uniWhenCompleteStage(

Executor e， BiConsumer<? super T， ? super Throwable> f) {

if (f == null) throw new NullPointerException();

CompletableFuture<T> d = new CompletableFuture<T>();

if (e != null || !d.uniWhenComplete(this， f， null)) {

// 当前CF未完成封装Completion

UniWhenComplete<T> c = new UniWhenComplete<T>(e， d， this， f);

// 压入栈顶stack变量

push(c);

c.tryFire(SYNC);

}

return d;

}

// WhenComplete封装，包含一个BiConsumer变量的UniCompletion

static final class UniWhenComplete<T> extends UniCompletion<T，T> {

BiConsumer<? super T， ? super Throwable> fn;

UniWhenComplete(Executor executor， CompletableFuture<T> dep，

CompletableFuture<T> src，

BiConsumer<? super T， ? super Throwable> fn) {

super(executor， dep， src); this.fn = fn;

}

final CompletableFuture<T> tryFire(int mode) {

CompletableFuture<T> d; CompletableFuture<T> a;

// 执行关联的CF的uniWhenComplete

if ((d = dep) == null || !d.uniWhenComplete(a = src， fn， mode > 0 ? null ： this))

return null;

dep = null; src = null; fn = null;

return d.postFire(a， mode);

}

}

// CF的WhenComplete完成回调

final boolean uniWhenComplete(CompletableFuture<T> a，

BiConsumer<? super T，? super Throwable> f，

UniWhenComplete<T> c) {

Object r; T t; Throwable x = null;

if (a == null || (r = a.result) == null || f == null)

return false;

if (result == null) {

try {

// 异步执行

if (c != null && !c.claim())

return false;

if (r instanceof AltResult) {

// 获取执行异常

x = ((AltResult)r).ex;

// AltResult的执行结果为null

t = null;

} else {

// 不为AltResult，设置t为执行结果

@SuppressWarnings("unchecked") T tr = (T) r;

t = tr;

}

// 调用f消费，执行结果t和执行异常

f.accept(t， x);

// 如果执行异常为空，那么将当前cf的结果完成为运行结果

if (x == null) {

internalComplete(r);

return true;

}

} catch (Throwable ex) {

if (x == null)

x = ex;

}

// 出现异常，设置为异常完成

completeThrowable(x， r);

}

return true;

}

接下来可以对CompletableFuture进行总结了：

1. 将每一个动作封装为CompletionStage，为CompletionStage的实现
2. 每个CompletableFuture都有一系列的同步方法和异步方法，命名为：xxx和xxxAsync
3. 每个CompletableFuture拥有一个stack执行栈，包含一系列的Completion，每个Completion和一个CompletableFuture关联
4. 初始通过CompletableFuture放入的根任务，通过封装为AsyncRun进行执行
5. 当每个CompletableFuture执行完毕后，如果当前stack执行栈不为空，那么依次回调栈内的Completion，Completion执行完毕后，如果这个Completion关联的CompletableFuture不为空，那么继续回调，直到所有Completion全部执行完毕

## 小结

本章详细介绍了不同线程池的实现，最基础的线程是ThreadPoolExecutor，其次是ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，实现了自己的任务调度算法，其中使用了DelayWorkQueue，其中采用小顶堆来实现，并且使用数组来加速任务的获取。读者需要注意的ThreadPoolExecutor的核心线程数，最大线程数，保活时间，线程工厂，阻塞队列，拒绝函数的原理，以及根据业务需要通过灵活配置它们来创建适应不同场景的线程池，这几个参数也是面试的重灾区，希望读者根据本章的逐行注释进行详细理解掌握。其次最复杂的线程池就是ForkJoinPool线程池，其中使用了工作窃取算法来加快任务的处理，充分利用了CPU资源，读者需要掌握其中的全局WorkQueue的奇数位和偶数位队列分别为内部工作队列和外部提交队列，核心的工作窃取方法scan以及帮助完成方法helpStealer的实现过程，这两个方法是难点和重点，希望读者可以静下心详细揣摩。最后读者需要理解掌握的类是CompletableFuture，这个类的出现让我们也可以在多线程中享受到Stream操作集合的快感，能让我们通过它对多线程的异步操作抽象为Stage，进行回调和实现。