

Gulimall

异步&线程池



一、线程回顾

1、初始化线程的 4 种方式

- 1)、继承 Thread
- 2)、实现 Runnable 接口
- 3)、实现 Callable 接口 + FutureTask （可以拿到返回结果，可以处理异常）
- 4)、线程池

方式 1 和方式 2：主进程无法获取线程的运算结果。不适合当前场景

方式 3：主进程可以获取线程的运算结果，但是不利于控制服务器中的线程资源。可以导致服务器资源耗尽。

方式 4：通过如下两种方式初始化线程池

```
Executors.newFixedThreadPool(3);  
//或者  
new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, TimeUnit unit,  
workQueue, threadFactory, handler);
```

通过线程池性能稳定，也可以获取执行结果，并捕获异常。但是，在业务复杂情况下，一个异步调用可能会依赖于另一个异步调用的执行结果。

2、线程池的七大参数

** @param corePoolSize the number of threads to keep in the pool, even
* if they are idle, unless {@code allowCoreThreadTimeOut} is set
池中一直保持的线程的数量，即使线程空闲。除非设置了allowCoreThreadTimeOut
* @param maximumPoolSize the maximum number of threads to allow in the
* pool*

池中允许的最大的线程数

** @param keepAliveTime when the number of threads is greater than
* the core, this is the maximum time that excess idle threads
* will wait for new tasks before terminating.*

当线程数大于核心线程数的时候，线程在最大多长时间没有接到新任务就会终止释放，最终线程池维持在corePoolSize大小

** @param unit the time unit for the {@code keepAliveTime} argument
时间单位*

** @param workQueue the queue to use for holding tasks before they are*

* *executed. This queue will hold only the {@code Runnable} tasks submitted by the {@code execute} method.*

阻塞队列，用来存储等待执行的任务，如果当前对线程的需求超过了 `corePoolSize` 大小，就会放在这里等待空闲线程执行。

* *@param threadFactory the factory to use when the executor creates a new thread*

创建线程的工厂，比如指定线程名等

* *@param handler the handler to use when execution is blocked*

* *because the thread bounds and queue capacities are reached*
拒绝策略，如果线程满了，线程池就会使用拒绝策略。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
{
    ...
}
```

运行流程：

- 1、线程池创建，准备好 `core` 数量的核心线程，准备接受任务
- 2、新的任务进来，用 `core` 准备好的空闲线程执行。
 - (1)、`core` 满了，就将再进来的任务放入阻塞队列中。空闲的 `core` 就会自己去阻塞队列获取任务执行
 - (2)、阻塞队列满了，就直接开新线程执行，最大只能开到 `max` 指定的数量
 - (3)、`max` 都执行好了。`Max-core` 数量空闲的线程会在 `keepAliveTime` 指定的时间后自动销毁。最终保持到 `core` 大小
 - (4)、如果线程数开到了 `max` 的数量，还有新任务进来，就会使用 `reject` 指定的拒绝策略进行处理
- 3、所有的线程创建都是由指定的 `factory` 创建的。

面试：

一个线程池 `core 7; max 20, queue: 50, 100` 并发进来怎么分配的；

先有 7 个能直接得到执行，接下来 50 个进入队列排队，在多开 13 个继续执行。现在 70 个被安排上了。剩下 30 个默认拒绝策略。

3、常见的 4 种线程池

- `newCachedThreadPool`
 - 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
- `newFixedThreadPool`
 - 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

- `newScheduledThreadPool`
 - 创建一个定长线程池，支持定时及周期性任务执行。
- `newSingleThreadExecutor`
 - 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

4、开发中为什么使用线程池

- 降低资源的消耗
 - 通过重复利用已经创建好的线程降低线程的创建和销毁带来的损耗
- 提高响应速度
 - 因为线程池中的线程数没有超过线程池的最大上限时，有的线程处于等待分配任务的状态，当任务来时无需创建新的线程就能执行
- 提高线程的可管理性
 - 线程池会根据当前系统特点对池内的线程进行优化处理，减少创建和销毁线程带来的系统开销。无限的创建和销毁线程不仅消耗系统资源，还降低系统的稳定性，使用线程池进行统一分配

二、CompletableFuture 异步编排

业务场景：

查询商品详情页的逻辑比较复杂，有些数据还需要远程调用，必然需要花费更多的时间。

```
// 1. 获取sku的基本信息    0.5s
// 2. 获取sku的图片信息    0.5s
// 3. 获取sku的促销信息    1s
// 4. 获取spu的所有销售属性  1s
// 5. 获取规格参数组及组下的规格参数  1.5s
// 6. spu详情              1s
```

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

那么，用户需要 5.5s 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 6 步操作，也许只需要 1.5s 即可完成响应。

`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。

虽然`Future`以及相关使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，为什么不能用观察者设计模式当计算结果完成及时通知监听者呢？

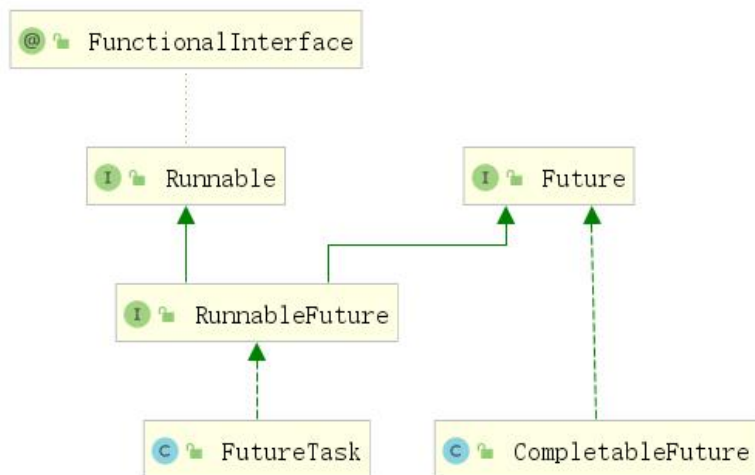
很多语言，比如 Node.js，采用回调的方式实现异步编程。Java 的一些框架，比如 Netty，自己扩展了 Java 的 `Future` 接口，提供了`addListener`等多个扩展方法；Google guava 也提供了通用的扩展 Future；Scala 也提供了简单易用且功能强大的 Future/Promise 异步编程模式。

作为正统的 Java 类库，是不是应该做点什么，加强一下自身库的功能呢？

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、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。

```
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)
```

- 1、runXxxx 都是没有返回结果的，supplyXxx 都是可以获取返回结果的
- 2、可以传入自定义的线程池，否则就用默认的线程池；

2、计算完成时回调方法

```
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, Executor executor);

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

whenComplete 可以处理正常和异常的计算结果，exceptionally 处理异常情况。

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。

whenCompleteAsync：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，也可能被同一个线程选中执行）

```
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());  
}  
}
```

3、handle 方法

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

和 complete 一样，可对结果做最后的处理（可处理异常），可改变返回值。

4、线程串行化方法

```
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 executor)  
  
public CompletionStage<Void> thenAccept(Consumer<? super T> action);  
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);  
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action,Executor  
executor);  
  
public CompletionStage<Void> thenRun(Runnable action);  
public CompletionStage<Void> thenRunAsync(Runnable action);  
public CompletionStage<Void> thenRunAsync(Runnable action,Executor executor);
```

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。

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。

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

带有 Async 默认是异步执行的。同之前。

以上都要前置任务成功完成。

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

5、两任务组合 - 都要完成

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

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

public <U,V> CompletableFuture<V> thenCombineAsync(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn, Executor executor);

public <U> CompletableFuture<Void> thenAcceptBoth(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action);

public <U> CompletableFuture<Void> thenAcceptBothAsync(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action);

public <U> CompletableFuture<Void> thenAcceptBothAsync(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action, Executor executor);

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

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

public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                  Runnable action,
                                                  Executor executor);
```

两个任务必须都完成，触发该任务。

thenCombine: 组合两个 future，获取两个 future 的返回结果，并返回当前任务的返回值

thenAcceptBoth: 组合两个 future，获取两个 future 任务的返回结果，然后处理任务，没有返回值。

runAfterBoth: 组合两个 future，不需要获取 future 的结果，只需两个 future 处理完任务后，处理该任务。

6、两任务组合 - 一个完成

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

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

public <U> CompletableFuture<U> applyToEitherAsync(
    CompletionStage<? extends T> other, Function<? super T, U> fn,
    Executor executor);

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

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

public CompletableFuture<Void> acceptEitherAsync(
    CompletionStage<? extends T> other, Consumer<? super T> action,
    Executor executor);
```

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

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

public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
    Runnable action,
    Executor executor);
```

当两个任务中，任意一个 **future** 任务完成的时候，执行任务。

applyToEither: 两个任务有一个执行完成，获取它的返回值，处理任务并有新的返回值。

acceptEither: 两个任务有一个执行完成，获取它的返回值，处理任务，没有新的返回值。

runAfterEither: 两个任务有一个执行完成，不需要获取 **future** 的结果，处理任务，也没有返回值。

7、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

