并发设计模式初探

Thursday, 1 September 2022   8:46 PM

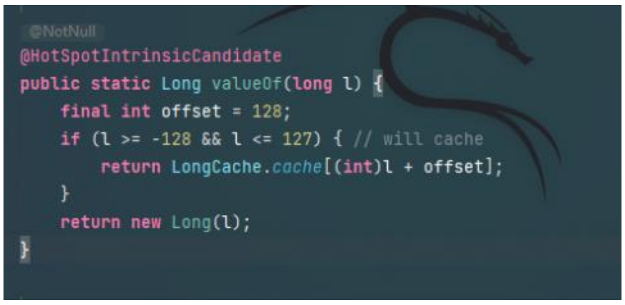
- 享元模式(Flyweight pattern)--->从并发数量带来的问题角度考虑：并发不仅仅是线程安全问题

  概念以及定义：

  wikipedia： A flyweight is an object that minimizes memory usage by sharing as much data as possible with other similar objects.

  体现：

  在JDK中 Boolean，Byte，Short，Integer，Long，Character 等包装类提供了 valueOf 方法，例如 Long 的valueOf 会缓存 -128~127 之间的 Long 对象，在这个范围之间会重用对象，大于这个范围，才会新建 Long 对象。



```java
@NotNull
@HotSpotIntrinsicCandidate
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= -128 && l <= 127) { // will cache
        return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}
```

【拓展】

- Byte, Short, Long 缓存的范围都是 -128~127
- Character 缓存的范围是 0~127
- Integer的默认范围是 -128~127最小值不能变，但最大值可以通过调整虚拟机参数 `-Djava.lang.Integer.IntegerCache.high` 来改变
- Boolean 缓存了 TRUE 和 FALSE

相同理念的一些实现：String字符串的常量池、链接池等等，统称"**池化**"思想。
这个模式在并发场景下是节省memory资源的利器！ 考虑到成熟的连接池(Jedis、Druid)代码过于复杂，短时间内不易"通透性"地把握，这里DIY一个简易版本的连接池。

Coding：

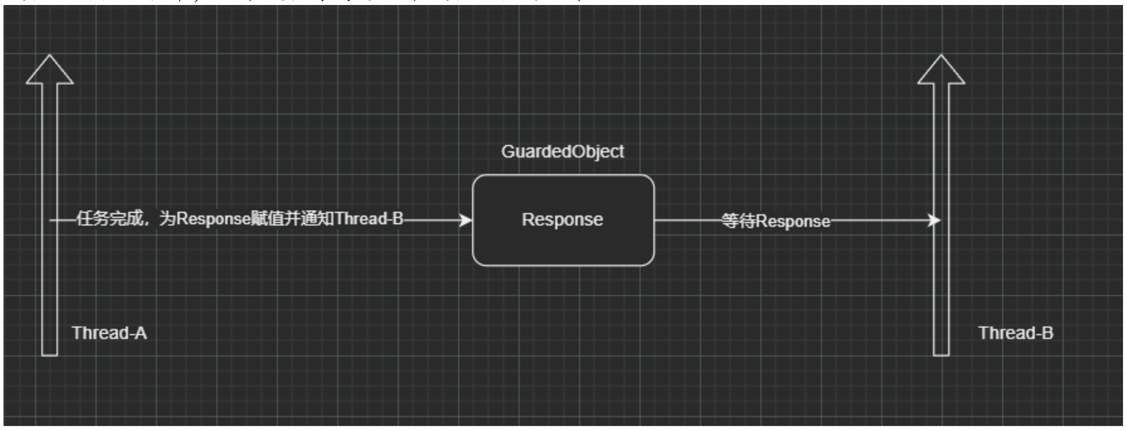
```java
class Pool {
    // 1. pool size
    private final int poolSize;
    // 2. connection array
    private Connection[] connections;
    // 3. connection state  0: available, 1: unavailable
    private AtomicIntegerArray states;

    // 4. <init>
    public Pool(int poolSize) {
        this.poolSize = poolSize;
        this.connections = new Connection[poolSize];
        this.states = new AtomicIntegerArray(new int[poolSize]);
        for (int i = 0; i < poolSize; i++) {
            connections[i] = new MockConnection("connection" + (i + 1));
        }
    }

    // 5. borrow connection
    public Connection borrow() {
        while (true) {
            for (int i = 0; i < poolSize; i++) {
                // try to get connection and sync state
                if (states.get(i) == 0) {
                    if (states.compareAndSet(i, 0, 1)) {
                        log.debug("borrow {}", connections[i]);
                        return connections[i];
                    }
                }
            }
            // if no connection is available, just wait
            synchronized (this) {
                try {
                    log.debug("wait...");
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    // 6. free and return connection
    public void free(Connection conn) {
        for (int i = 0; i < poolSize; i++) {
            if (connections[i] == conn) {
                states.set(i, 0);
                synchronized (this) {
                    log.debug("free {}", conn);
                    this.notifyAll();
                }
                break;
            }
        }
    }
```

```
        // Connection
    class MockConnection implements Connection {
        public MockConnection(String connectionName) {
            // ...
        }
    }
}
```

➢ 同步模式之保护性暂停(Guarded Suspension)

概念以及定义:

线程的通信过程中，一个线程等待另一个线程的执行结果



使用背景以及要点:

有一个结果需要从一个线程传递到另一个线程，可以让他们关联同一个GuardedObject
如果有结果不断从一个线程到另一个线程那么可以使用消息队列（见生产者/消费者）

体现:

JDK 中，join方法的实现、Future类的实现，采用的就是此模式。(因为要等待另一方的结果，因此归类到同步模式)

Coding

➢ ---

template coding(模板代码):

```
class GuardedObject {

    private final Object lock = new Object();
    private Object response;

    public Object get() {
        synchronized (lock) {
            // condition isn't satisfied
            while (response == null) {
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            return response;
        }
    }

    public void complete(Object response) {
        synchronized (lock) {
            // when the condition is satisfied, notify other threads
            this.response = response;
            lock.notifyAll();
        }
    }
}
```

practice coding:

```
public static void main(String[] args) {
    GuardedObject guardedObject = new GuardedObject();
    new Thread(() -> {
        try {
            // Thread S is working...
            List<String> response = download();
            log.debug("download complete...");
            guardedObject.complete(response);
        } catch (IOException e) {
            e.printStackTrace();
        }
    },"S").start();

    log.debug("waiting...");
    // Thread main will block and wait
    Object response = guardedObject.get();
    log.debug("get response: [{}] lines", ((List<String>) response).size());
}
```

---

template upgrade coding with time record(升级计时功能)

```
class GuardedObjectV2 {
    private final Object lock = new Object();
    private Object response;
    public Object get(long millis) {
        synchronized (lock) {
            // record the start time
            long begin = System.currentTimeMillis();
            // duration
```

practice coding

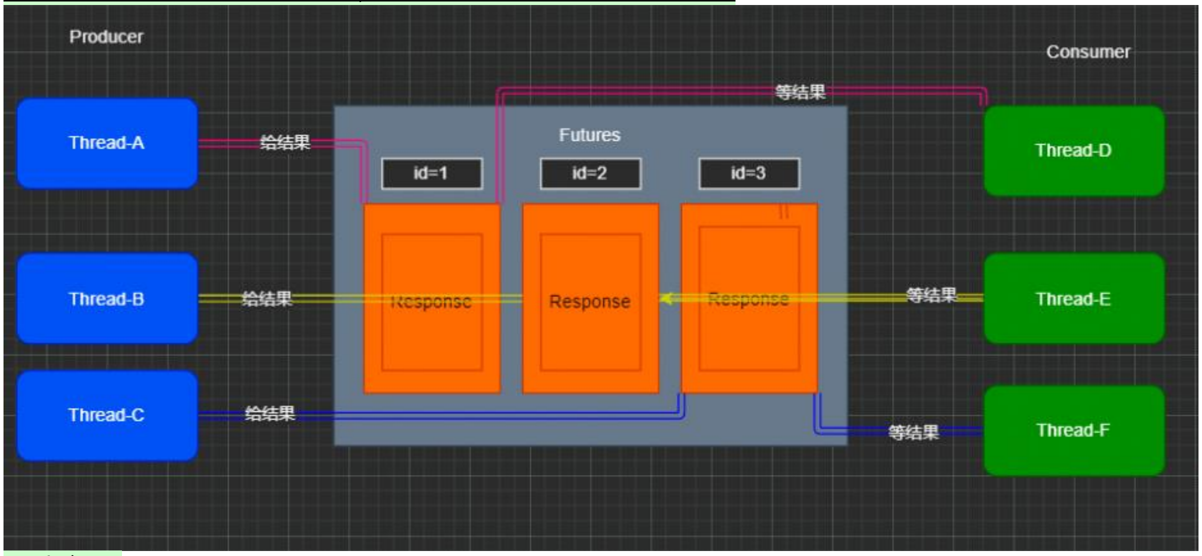
```
public static void main(String[] args) {
    GuardedObjectV2 v2 = new GuardedObjectV2();
    new Thread(() -> {
        sleep(1);
        v2.complete(null);
        sleep(1);
        v2.complete(Arrays.asList("a", "b", "c"));
    }).start();
    public Object get(long millis) {
        synchronized (lock) {
            // record the start time
            long begin = System.currentTimeMillis();
            // duration
            long timePassed = 0;
            while (response == null) {
                // rest time to wait
                long waitTime = millis - timePassed;
                log.debug("waitTime: {}", waitTime);
                if (waitTime <= 0) {
                    log.debug("break...");
                    break;
                }
                try {
                    lock.wait(waitTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // if the thread is woke up, just logging
                timePassed = System.currentTimeMillis() - begin;
                log.debug("timePassed: {}, object is null {}",
                        timePassed, response == null);
            }
            return response;
        }
    }
    public void complete(Object response) {
        synchronized (lock) {
            // condition is satisfied, notify all blocking threads
            this.response = response;
            log.debug("notify...");
            lock.notifyAll();
        }
    }
}
```

```
        sleep(1);
        v2.complete(null);
        sleep(1);
        v2.complete(Arrays.asList("a", "b", "c"));
    }).start();
    Object response = v2.get(2500);
    if (response != null) {
        log.debug("get response: [{}] lines",
((List<String>) response).size());
    } else {
        log.debug("can't get response");
    }
}
```

---

coding template with multi task(升级多任务管理)

如果需要在多个类之间使用 GuardedObject 对象，作为参数传递不是很方便，因此设计一个用来解耦的中间类，这样不仅能够解耦【结果等待者】和【结果生产者】，还能够同时支持多个任务的管理。



场景举例：

多个邮递员给多个居民家的邮箱投递信件

Guarded Object(新增id用来标识)

```
class GuardedObject {
    private int id;
    private Object response;
    public GuardedObject(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public Object get(long timeout) {
        synchronized (this) {
            long begin = System.currentTimeMillis();
            long passedTime = 0;
            while (response == null) {
                long waitTime = timeout - passedTime;
                if (timeout - passedTime <= 0) {
                    break;
                }
                try {
                    this.wait(waitTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                passedTime = System.currentTimeMillis() - begin;
            }
            return response;
        }
    }
    public void complete(Object response) {
        synchronized (this) {
            this.response = response;
            this.notifyAll();
        }
    }
}
```

邮箱集合：

```
class Mailboxes {
    private static Map<Integer, GuardedObject> boxes = new Hashtable<>();
    private static int id = 1;

    private static synchronized int generateId() {
        return id++;
    }
    public static GuardedObject getGuardedObject(int id) {
        return boxes.remove(id);
    }
    public static GuardedObject createGuardedObject() {
        GuardedObject go = new GuardedObject(generateId());
        boxes.put(go.getId(), go);
        return go;
    }
    public static Set<Integer> getIds() {
        return boxes.keySet();
    }
}
```

居民

```
class Resident extends Thread{
    @Override
    public void run() {
        GuardedObject guardedObject = Mailboxes.createGuardedObject();
        log.debug("开始收信 id:{}", guardedObject.getId());
        Object mail = guardedObject.get(5000);
        log.debug("收到信 id:{}, 内容:{}", guardedObject.getId(), mail);
    }
}
```

邮递员

```
class Postman extends Thread {
    private int id;
    private String mail;
    public Postman(int id, String mail) {
        this.id = id;
        this.mail = mail;
    }
    @Override
    public void run() {
        GuardedObject guardedObject = Mailboxes.getGuardedObject(id);
        log.debug("送信 id:{}, 内容:{}", id, mail);
        guardedObject.complete(mail);
    }
}
```

测试

```
public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < 3; i++) {
        new Resident().start();
    }
    Sleeper.sleep(1);
    for (Integer id : Mailboxes.getIds()) {
        new Postman(id, "内容" + id).start();
    }
}
//10:35:05.689 c.Resident [Thread-1] - 开始收信 id:3
//10:35:05.689 c.Resident [Thread-2] - 开始收信 id:1
//10:35:05.689 c.Resident [Thread-0] - 开始收信 id:2
//10:35:06.688 c.Postman [Thread-4] - 送信 id:2, 内容:内容2
//10:35:06.688 c.Postman [Thread-5] - 送信 id:1, 内容:内容1
//10:35:06.688 c.Resident [Thread-0] - 收到信 id:2, 内容:内容2
//10:35:06.688 c.Resident [Thread-2] - 收到信 id:1, 内容:内容1
//10:35:06.688 c.Postman [Thread-3] - 送信 id:3, 内容:内容3
//10:35:06.689 c.Resident [Thread-1] - 收到信 id:3, 内容:内容3
```

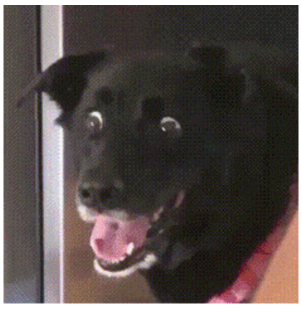
➢ 同步模式之犹豫模式(Balking)

概念以及定义：

Balking (犹豫)模式用在一个线程发现另一个线程或本线程已经做了某一件相同的事，那么本线程就无需再做了，直接结束返回。

使用背景：

经常用来实现线程安全的单例。可以说是另一种形式的幂等。
如果有一天你需要在Openshift上对一个pod进行scale， 由0到1…点击up之后发现页面卡主，于是你又点击了一次up,然后…?
如果有一天你在上海消费券申请页面点了一次申请后卡住了...?
如果有一个监控程序点击后就启动监控，虽然没卡主，但是你还是点击了多次，然后...?



多次点击别担心，单例模式照顾您。

如下模板代码：

```
public final class Singleton {
    private static Singleton INSTANCE = null;
    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if (INSTANCE != null) {
            return INSTANCE;
        }
        INSTANCE = new Singleton();
        return INSTANCE;
    }
}
```

Coding：

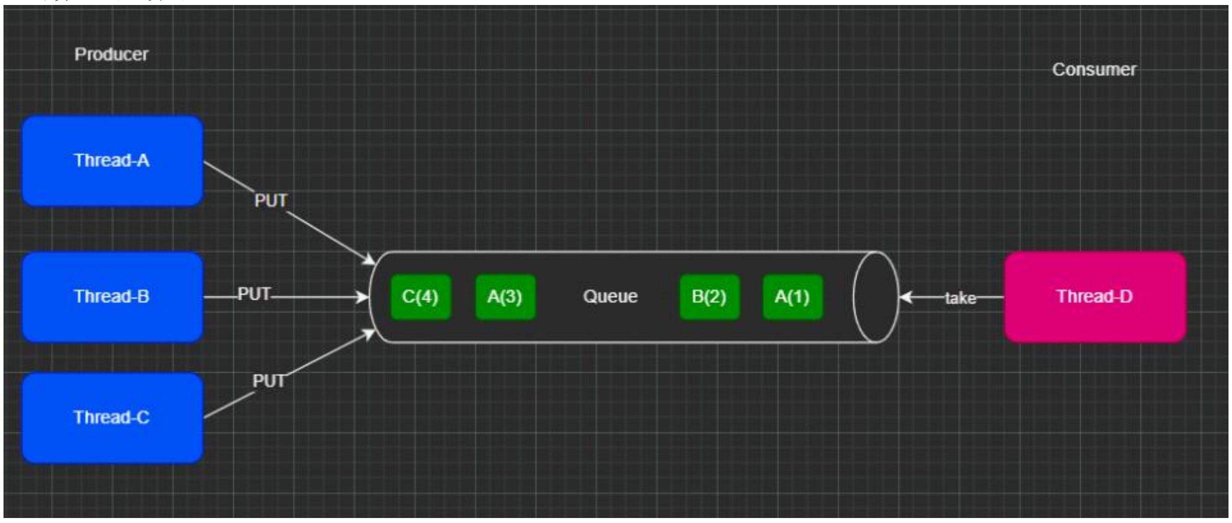
就以监控程序为例：

```
public class MonitorService {
    // is already starting
    private volatile boolean starting;
    public void start() {
        log.info("尝试启动监控线程...");
        synchronized (this) {
            if (starting) {
                return;
            }
            starting = true;
        }

        // 真正启动监控...
    }
}
```

➢ 异步模式之生产者&消费者

概念以及定义：

异步化结构，与前面的保护性暂停中的 GuardObject 不同，不需要产生结果和消费结果的线程一一对应消费队列可以用来平衡生产和消费的线程资源。



要点以及体现：

生产者仅负责产生结果数据，不关心数据该如何处理，而消费者专心处理结果数据

消息队列是有容量限制的，满时不会再加入数据，空时不会再消耗数据

JDK 中各种阻塞队列，采用的就是这种模式

Coding：

我们来DIY一个简单地消息队列，用生产者和消费者来模拟：

消息实体

```
class Message {
    private int id;
    private Object message;
    public Message(int id, Object message) {
        this.id = id;
        this.message = message;
    }
    public int getId() {
        return id;
    }
    public Object getMessage() {
        return message;
    }
}
```

消息队列

```
class MessageQueue {
    private LinkedList<Message> queue;
    private int capacity;

    public MessageQueue(int capacity) {
        this.capacity = capacity;
        queue = new LinkedList<>();
    }

    public Message take() {
        synchronized (queue) {
            while (queue.isEmpty()) {
                log.debug("没货了, wait");
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            Message message = queue.removeFirst();
            queue.notifyAll();
            return message;
        }
    }

    public void put(Message message) {
        synchronized (queue) {
            while (queue.size() == capacity) {
                log.debug("库存已达上限, wait");
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            queue.addLast(message);
            queue.notifyAll();
        }
    }
}
```

运行代码

```
MessageQueue messageQueue = new MessageQueue(2);
// 4个生产者线程进行生产，将下载结果作为message内容放入队列
for (int i = 0; i < 4; i++) {
    int id = i;
    new Thread(() -> {
        try {
            log.debug("download...");
            List<String> response = Downloader.download();
            log.debug("try put message({})", id);
            messageQueue.put(new Message(id, response));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }, "生产者" + i).start();
}
// 1个消费者线程
new Thread(() -> {
    while (true) {
        Message message = messageQueue.take();
        List<String> response = (List<String>) message.getMessage();
        log.debug("take message({}): [{}] lines", message.getId(), response.size());
    }
}, "消费者").start();

//10:48:38.070 [生产者3] c.TestProducerConsumer - download...
//10:48:38.070 [生产者0] c.TestProducerConsumer - download...
//10:48:38.070 [消费者] c.MessageQueue - 没货了, wait
//10:48:38.070 [生产者1] c.TestProducerConsumer - download...
//10:48:38.070 [生产者2] c.TestProducerConsumer - download...
//10:48:41.236 [生产者1] c.TestProducerConsumer - try put message(1)
//10:48:41.237 [生产者2] c.TestProducerConsumer - try put message(2)
//10:48:41.236 [生产者0] c.TestProducerConsumer - try put message(0)
//10:48:41.237 [生产者3] c.TestProducerConsumer - try put message(3)
//10:48:41.239 [生产者2] c.MessageQueue - 库存已达上限, wait
//10:48:41.240 [生产者1] c.MessageQueue - 库存已达上限, wait
//10:48:41.240 [消费者] c.TestProducerConsumer - take message(0): [3] lines
//10:48:41.240 [生产者2] c.MessageQueue - 库存已达上限, wait
//10:48:41.240 [消费者] c.TestProducerConsumer - take message(3): [3] lines
//10:48:41.240 [消费者] c.TestProducerConsumer - take message(1): [3] lines
//10:48:41.240 [消费者] c.TestProducerConsumer - take message(2): [3] lines
//10:48:41.240 [消费者] c.MessageQueue - 没货了, wait
```

➢ 想要精通并发编程模式？还有如下这些：

同步模式之顺序控制

  固定顺序

  交替输出

两阶段终止模式

  …………

***模式

未完待续...