

# 模式篇

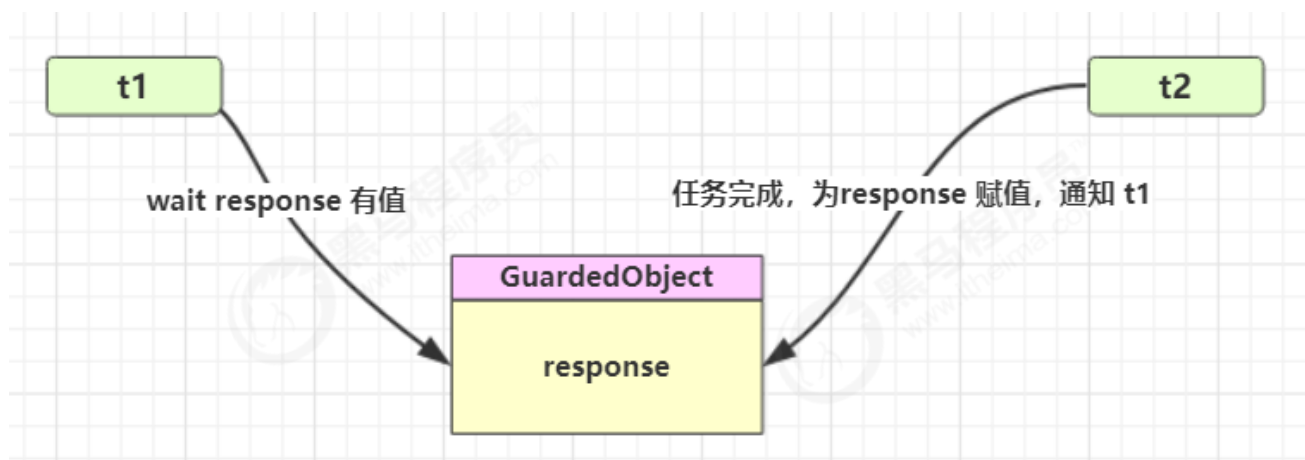
## 同步模式之保护性暂停

### 1. 定义

即 Guarded Suspension，用在一个线程等待另一个线程的执行结果

要点

- 有一个结果需要从一个线程传递到另一个线程，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）
- JDK 中，join 的实现、Future 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



### 2. 实现

```
class GuardedObject {  
  
    private Object response;  
    private final Object lock = new Object();  
  
    public Object get() {  
        synchronized (lock) {  
            // 条件不满足则等待  
            while (response == null) {  
                try {  
                    lock.wait();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
            return response;  
        }  
    }  
}
```

```
}

public void complete(Object response) {
    synchronized (lock) {
        // 条件满足，通知等待线程
        this.response = response;
        lock.notifyAll();
    }
}
}
```

## \* 应用

一个线程等待另一个线程的执行结果

```
public static void main(String[] args) {
    GuardedObject guardedObject = new GuardedObject();
    new Thread(() -> {
        try {
            // 子线程执行下载
            List<String> response = download();
            log.debug("download complete...");
            guardedObject.complete(response);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }).start();

    log.debug("waiting...");
    // 主线程阻塞等待
    Object response = guardedObject.get();
    log.debug("get response: [{}] lines", ((List<String>) response).size());
}
```

执行结果

```
08:42:18.568 [main] c.TestGuardedObject - waiting...
08:42:23.312 [Thread-0] c.TestGuardedObject - download complete...
08:42:23.312 [main] c.TestGuardedObject - get response: [3] lines
```

## 3. 带超时版 GuardedObject

如果要控制超时时间呢

```
class GuardedObjectV2 {

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



```
public Object get(long millis) {
    synchronized (lock) {
        // 1) 记录最初时间
        long begin = System.currentTimeMillis();
        // 2) 已经经历的时间
        long timePassed = 0;
        while (response == null) {
            // 4) 假设 millis 是 1000, 结果在 400 时唤醒了, 那么还有 600 要等
            long waitTime = millis - timePassed;
            log.debug("waitTime: {}", waitTime);
            if (waitTime <= 0) {
                log.debug("break...");
                break;
            }
            try {
                lock.wait(waitTime);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 3) 如果提前被唤醒, 这时已经经历的时间假设为 400
            timePassed = System.currentTimeMillis() - begin;
            log.debug("timePassed: {}, object is null {}",
                timePassed, response == null);
        }
        return response;
    }
}

public void complete(Object response) {
    synchronized (lock) {
        // 条件满足, 通知等待线程
        this.response = response;
        log.debug("notify...");
        lock.notifyAll();
    }
}
}
```

测试, 没有超时

```
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();

    Object response = v2.get(2500);
    if (response != null) {

        log.debug("get response: [{}] lines", ((List<String>) response).size());
    }
}
```

```
} else {  
    log.debug("can't get response");  
}  
}
```

输出

```
08:49:39.917 [main] c.GuardedObjectV2 - waitTime: 2500  
08:49:40.917 [Thread-0] c.GuardedObjectV2 - notify...  
08:49:40.917 [main] c.GuardedObjectV2 - timePassed: 1003, object is null true  
08:49:40.917 [main] c.GuardedObjectV2 - waitTime: 1497  
08:49:41.918 [Thread-0] c.GuardedObjectV2 - notify...  
08:49:41.918 [main] c.GuardedObjectV2 - timePassed: 2004, object is null false  
08:49:41.918 [main] c.TestGuardedObjectV2 - get response: [3] lines
```

测试，超时

```
// 等待时间不足  
List<String> lines = v2.get(1500);
```

输出

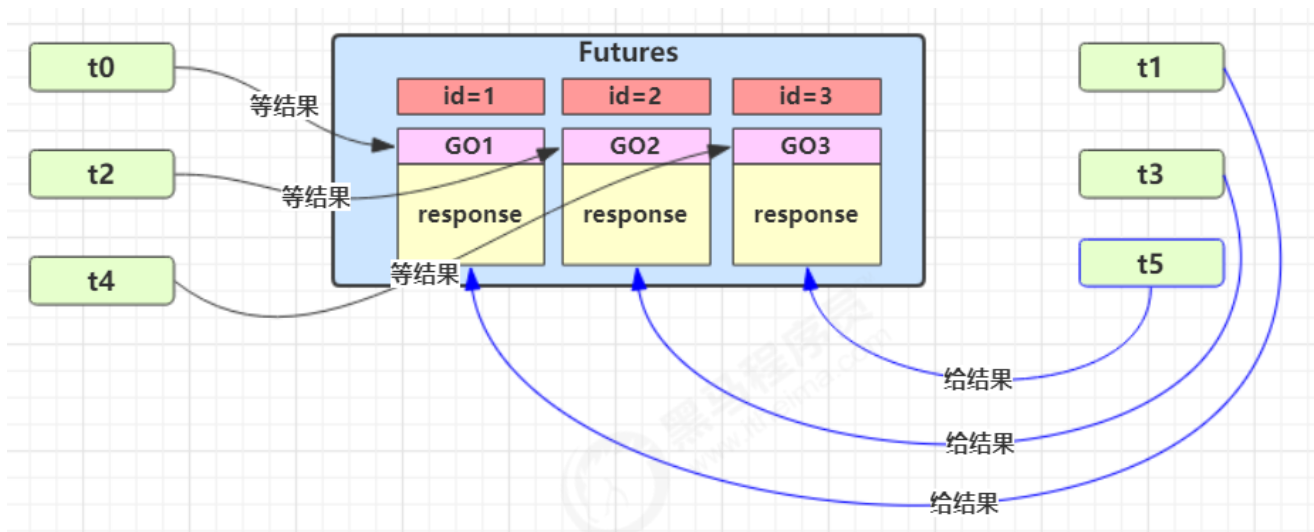
```
08:47:54.963 [main] c.GuardedObjectV2 - waitTime: 1500  
08:47:55.963 [Thread-0] c.GuardedObjectV2 - notify...  
08:47:55.963 [main] c.GuardedObjectV2 - timePassed: 1002, object is null true  
08:47:55.963 [main] c.GuardedObjectV2 - waitTime: 498  
08:47:56.461 [main] c.GuardedObjectV2 - timePassed: 1500, object is null true  
08:47:56.461 [main] c.GuardedObjectV2 - waitTime: 0  
08:47:56.461 [main] c.GuardedObjectV2 - break...  
08:47:56.461 [main] c.TestGuardedObjectV2 - can't get response  
08:47:56.963 [Thread-0] c.GuardedObjectV2 - notify...
```

## \* 原理之 join

### 4. 多任务版 GuardedObject

图中 Futures 就好比居民楼一层的信箱（每个信箱有房间编号），左侧的 t0，t2，t4 就好比等待邮件的居民，右侧的 t1，t3，t5 就好比邮递员

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



新增 id 用来标识 Guarded Object

```
class GuardedObject {  
  
    // 标识 Guarded Object  
    private int id;  
  
    public GuardedObject(int id) {  
        this.id = id;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    // 结果  
    private Object response;  
  
    // 获取结果  
    // timeout 表示要等待多久 2000  
    public Object get(long timeout) {  
        synchronized (this) {  
            // 开始时间 15:00:00  
            long begin = System.currentTimeMillis();  
            // 经历的时间  
            long passedTime = 0;  
            while (response == null) {  
                // 这一轮循环应该等待的时间  
                long waitTime = timeout - passedTime;  
                // 经历的时间超过了最大等待时间时，退出循环  
                if (timeout - passedTime <= 0) {  
                    break;  
                }  
                try {  
                    this.wait(waitTime); // 虚假唤醒 15:00:01  
                } catch (InterruptedException e) {  
  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

```
    }  
    // 求得经历时间  
    passedTime = System.currentTimeMillis() - begin; // 15:00:02 1s  
    }  
    return response;  
    }  
}  
  
// 产生结果  
public void complete(Object response) {  
    synchronized (this) {  
        // 给结果成员变量赋值  
        this.response = response;  
        this.notifyAll();  
    }  
}  
}
```

### 中间解耦类

```
class Mailboxes {  
    private static Map<Integer, GuardedObject> boxes = new Hashtable<>();  
  
    private static int id = 1;  
    // 产生唯一 id  
    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();  
    }  
}
```

### 业务相关类



```
class People extends Thread{
    @Override
    public void run() {
        // 收信
        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);
    }
}
```

```
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 People().start();
    }
    Sleeper.sleep(1);
    for (Integer id : Mailboxes.getIds()) {
        new Postman(id, "内容" + id).start();
    }
}
```

## 某次运行结果

```
10:35:05.689 c.People [Thread-1] - 开始收信 id:3
10:35:05.689 c.People [Thread-2] - 开始收信 id:1
10:35:05.689 c.People [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.People [Thread-0] - 收到信 id:2, 内容:内容2
10:35:06.688 c.People [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.People [Thread-1] - 收到信 id:3, 内容:内容3
```

## 同步模式之 Balking

### 1. 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事，那么本线程就无需再做了，直接结束返回

### 2. 实现

例如：

```
public class MonitorService {

    // 用来表示是否已经有线程已经在执行启动了
    private volatile boolean starting;

    public void start() {
        log.info("尝试启动监控线程...");
        synchronized (this) {
            if (starting) {
                return;
            }
            starting = true;
        }

        // 真正启动监控线程...
    }
}
```

当前端页面多次点击按钮调用 start 时

输出



```
[http-nio-8080-exec-1] cn.itcast.monitor.service.MonitorService - 该监控线程已启动?(false)
[http-nio-8080-exec-1] cn.itcast.monitor.service.MonitorService - 监控线程已启动...
[http-nio-8080-exec-2] cn.itcast.monitor.service.MonitorService - 该监控线程已启动?(true)
[http-nio-8080-exec-3] cn.itcast.monitor.service.MonitorService - 该监控线程已启动?(true)
[http-nio-8080-exec-4] cn.itcast.monitor.service.MonitorService - 该监控线程已启动?(true)
```

它还经常用来实现线程安全的单例

```
public final class Singleton {
    private Singleton() {
    }

    private static Singleton INSTANCE = null;

    public static synchronized Singleton getInstance() {
        if (INSTANCE != null) {
            return INSTANCE;
        }

        INSTANCE = new Singleton();
        return INSTANCE;
    }
}
```

对比一下保护性暂停模式：保护性暂停模式用在一个线程等待另一个线程的执行结果，当条件不满足时线程等待。

## 同步模式之顺序控制

### 1. 固定运行顺序

比如，必须先 2 后 1 打印

#### 1.1 wait notify 版

```
// 用来同步的对象
static Object obj = new Object();
// t2 运行标记，代表 t2 是否执行过
static boolean t2runed = false;

public static void main(String[] args) {

    Thread t1 = new Thread(() -> {
        synchronized (obj) {
            // 如果 t2 没有执行过
            while (!t2runed) {
                try {
                    // t1 先等一会
                    obj.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    });
}
```

```
    }  
    }  
    }  
    System.out.println(1);  
});  
  
Thread t2 = new Thread(() -> {  
    System.out.println(2);  
    synchronized (obj) {  
        // 修改运行标记  
        t2runed = true;  
        // 通知 obj 上等待的线程 (可能有多条, 因此需要用 notifyAll)  
        obj.notifyAll();  
    }  
});  
  
t1.start();  
t2.start();  
}
```

## 1.2 Park Unpark 版

可以看到，实现上很麻烦：

- 首先，需要保证先 wait 再 notify，否则 wait 线程永远得不到唤醒。因此使用了『运行标记』来判断该不该 wait
- 第二，如果有些干扰线程错误地 notify 了 wait 线程，条件不满足时还要重新等待，使用了 while 循环来解决此问题
- 最后，唤醒对象上的 wait 线程需要使用 notifyAll，因为『同步对象』上的等待线程可能不止一个

可以使用 LockSupport 类的 park 和 unpark 来简化上面的题目：

```
Thread t1 = new Thread(() -> {  
    try { Thread.sleep(1000); } catch (InterruptedException e) { }  
    // 当没有『许可』时，当前线程暂停运行；有『许可』时，用掉这个『许可』，当前线程恢复运行  
    LockSupport.park();  
    System.out.println("1");  
});  
  
Thread t2 = new Thread(() -> {  
    System.out.println("2");  
    // 给线程 t1 发放『许可』（多次连续调用 unpark 只会发放一个『许可』）  
    LockSupport.unpark(t1);  
});  
  
t1.start();  
t2.start();
```

park 和 unpark 方法比较灵活，他俩谁先调用，谁后调用无所谓。并且是以线程为单位进行『暂停』和『恢复』，不需要『同步对象』和『运行标记』

## 2. 交替输出

线程 1 输出 a 5 次，线程 2 输出 b 5 次，线程 3 输出 c 5 次。现在要求输出 abcabcabcabcabc 怎么实现

## 2.1 wait notify 版

```
class SyncWaitNotify {
    private int flag;
    private int loopNumber;

    public SyncWaitNotify(int flag, int loopNumber) {
        this.flag = flag;
        this.loopNumber = loopNumber;
    }

    public void print(int waitFlag, int nextFlag, String str) {
        for (int i = 0; i < loopNumber; i++) {
            synchronized (this) {
                while (this.flag != waitFlag) {
                    try {
                        this.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.print(str);
                flag = nextFlag;
                this.notifyAll();
            }
        }
    }
}
```

```
SyncWaitNotify syncWaitNotify = new SyncWaitNotify(1, 5);
new Thread(() -> {
    syncWaitNotify.print(1, 2, "a");
}).start();
new Thread(() -> {
    syncWaitNotify.print(2, 3, "b");
}).start();
new Thread(() -> {
    syncWaitNotify.print(3, 1, "c");
}).start();
```

## 2.2 Lock 条件变量版

```
class AwaitSignal extends ReentrantLock {
    public void start(Condition first) {
        this.lock();
    }
}
```



```
        try {
            log.debug("start");
            first.signal();
        } finally {
            this.unlock();
        }
    }

    public void print(String str, Condition current, Condition next) {
        for (int i = 0; i < loopNumber; i++) {
            this.lock();
            try {
                current.await();
                log.debug(str);
                next.signal();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                this.unlock();
            }
        }
    }

    // 循环次数
    private int loopNumber;

    public AwaitSignal(int loopNumber) {
        this.loopNumber = loopNumber;
    }
}
```

```
AwaitSignal as = new AwaitSignal(5);
Condition aWaitSet = as.newCondition();
Condition bWaitSet = as.newCondition();
Condition cWaitSet = as.newCondition();

new Thread(() -> {
    as.print("a", aWaitSet, bWaitSet);
}).start();
new Thread(() -> {
    as.print("b", bWaitSet, cWaitSet);
}).start();
new Thread(() -> {
    as.print("c", cWaitSet, aWaitSet);
}).start();

as.start(aWaitSet);
```

## 注意

该实现没有考虑 a, b, c 线程都就绪再开始

## 2.3 Park Unpark 版

```
class SyncPark {
    private int loopNumber;
    private Thread[] threads;

    public SyncPark(int loopNumber) {
        this.loopNumber = loopNumber;
    }

    public void setThreads(Thread... threads) {
        this.threads = threads;
    }

    public void print(String str) {
        for (int i = 0; i < loopNumber; i++) {
            LockSupport.park();
            System.out.print(str);
            LockSupport.unpark(nextThread());
        }
    }

    private Thread nextThread() {
        Thread current = Thread.currentThread();
        int index = 0;
        for (int i = 0; i < threads.length; i++) {
            if(threads[i] == current) {
                index = i;
                break;
            }
        }
        if(index < threads.length - 1) {
            return threads[index+1];
        } else {
            return threads[0];
        }
    }

    public void start() {
        for (Thread thread : threads) {
            thread.start();
        }
        LockSupport.unpark(threads[0]);
    }
}
```

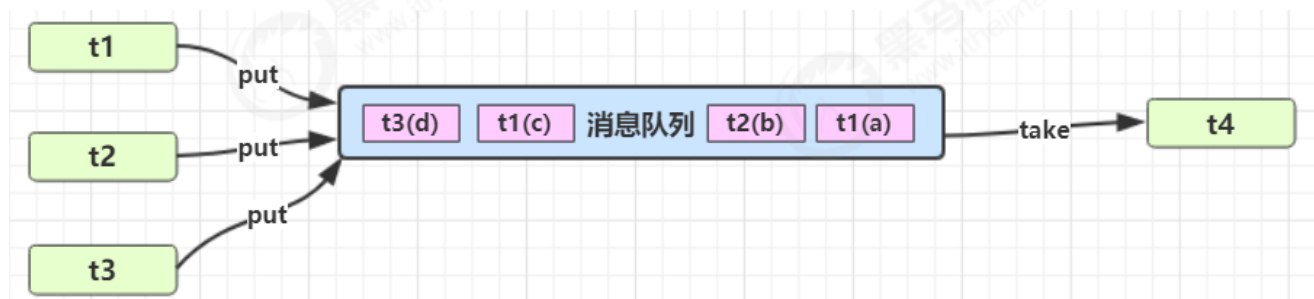
```
SyncPark syncPark = new SyncPark(5);
Thread t1 = new Thread(() -> {
    syncPark.print("a");
});
Thread t2 = new Thread(() -> {
    syncPark.print("b");
});
Thread t3 = new Thread(() -> {
    syncPark.print("c\n");
});
syncPark.setThreads(t1, t2, t3);
syncPark.start();
```

## 异步模式之生产者/消费者

### 1. 定义

#### 要点

- 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——对应
- 消费队列可以用来平衡生产和消费的线程资源
- 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



### 2. 实现

```
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;
    }
}
```



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

## \* 应用

```
MessageQueue messageQueue = new MessageQueue(2);  
// 4 个生产者线程，下载任务  
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，处理结果
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();
```

某次运行结果

```
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
```

结果解读

## 异步模式之工作线程

### 1. 定义

让有限的工作线程 (Worker Thread) 来轮流异步处理无限多的任务。也可以将其归类为分工模式，它的典型实现就是线程池，也体现了经典设计模式中的享元模式。



例如，海底捞的服务员（线程），轮流处理每位客人的点餐（任务），如果为每位客人都配一名专属的服务员，那么成本就太高了（对比另一种多线程设计模式：Thread-Per-Message）

注意，不同任务类型应该使用不同的线程池，这样能够避免饥饿，并能提升效率

例如，如果一个餐馆的工人既要招呼客人（任务类型A），又要到后厨做菜（任务类型B）显然效率不咋地，分成服务员（线程池A）与厨师（线程池B）更为合理，当然你能想到更细致的分工

## 2. 饥饿

固定大小线程池会有饥饿现象

- 两个工人是同一个线程池中的两个线程
- 他们要做的事情是：为客人点餐和到后厨做菜，这是两个阶段的工作
  - 客人点餐：必须先点完餐，等菜做好，上菜，在此期间处理点餐的工人必须等待
  - 后厨做菜：没啥说的，做就是了
- 比如工人A处理了点餐任务，接下来它要等着工人B把菜做好，然后上菜，他俩也配合的蛮好
- 但现在同时来了两个客人，这个时候工人A和工人B都去处理点餐了，这时没人做饭了，饥饿

```
public class TestDeadLock {  
  
    static final List<String> MENU = Arrays.asList("地三鲜", "宫保鸡丁", "辣子鸡丁", "烤鸡翅");  
    static Random RANDOM = new Random();  
    static String cooking() {  
        return MENU.get(RANDOM.nextInt(MENU.size()));  
    }  
    public static void main(String[] args) {  
        ExecutorService executorService = Executors.newFixedThreadPool(2);  
  
        executorService.execute(() -> {  
            log.debug("处理点餐...");  
            Future<String> f = executorService.submit(() -> {  
                log.debug("做菜");  
                return cooking();  
            });  
            try {  
                log.debug("上菜: {}", f.get());  
            } catch (InterruptedException | ExecutionException e) {  
                e.printStackTrace();  
            }  
        });  
  
        /*executorService.execute(() -> {  
            log.debug("处理点餐...");  
            Future<String> f = executorService.submit(() -> {  
                log.debug("做菜");  
                return cooking();  
            });  
            try {  
                log.debug("上菜: {}", f.get());  
            } catch (InterruptedException | ExecutionException e) {  
                e.printStackTrace();  
            }  
        });  
    }  
}
```

```
        e.printStackTrace();
    }
});*/

}
```

## 输出

```
17:21:27.883 c.TestDeadLock [pool-1-thread-1] - 处理点餐...
17:21:27.891 c.TestDeadLock [pool-1-thread-2] - 做菜
17:21:27.891 c.TestDeadLock [pool-1-thread-1] - 上菜: 烤鸡翅
```

当注释取消后，可能的输出

```
17:08:41.339 c.TestDeadLock [pool-1-thread-2] - 处理点餐...
17:08:41.339 c.TestDeadLock [pool-1-thread-1] - 处理点餐...
```

解决方法可以增加线程池的大小，不过不是根本解决方案，还是前面提到的，不同的任务类型，采用不同的线程池，例如：

```
public class TestDeadLock {

    static final List<String> MENU = Arrays.asList("地三鲜", "宫保鸡丁", "辣子鸡丁", "烤鸡翅");
    static Random RANDOM = new Random();
    static String cooking() {
        return MENU.get(RANDOM.nextInt(MENU.size()));
    }
    public static void main(String[] args) {
        ExecutorService waiterPool = Executors.newFixedThreadPool(1);
        ExecutorService cookPool = Executors.newFixedThreadPool(1);

        waiterPool.execute(() -> {
            log.debug("处理点餐...");
            Future<String> f = cookPool.submit(() -> {
                log.debug("做菜");
                return cooking();
            });
            try {
                log.debug("上菜: {}", f.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        });
        waiterPool.execute(() -> {
            log.debug("处理点餐...");
            Future<String> f = cookPool.submit(() -> {
                log.debug("做菜");
                return cooking();
            });
            try {
```

```
        log.debug("上菜: {}", f.get());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
});

}
```

输出

```
17:25:14.626 c.TestDeadLock [pool-1-thread-1] - 处理点餐...
17:25:14.630 c.TestDeadLock [pool-2-thread-1] - 做菜
17:25:14.631 c.TestDeadLock [pool-1-thread-1] - 上菜: 地三鲜
17:25:14.632 c.TestDeadLock [pool-1-thread-1] - 处理点餐...
17:25:14.632 c.TestDeadLock [pool-2-thread-1] - 做菜
17:25:14.632 c.TestDeadLock [pool-1-thread-1] - 上菜: 辣子鸡丁
```

### 3. 创建多少线程池合适

- 过小会导致程序不能充分地利用系统资源、容易导致饥饿
- 过大会导致更多的线程上下文切换，占用更多内存

#### 3.1 CPU 密集型运算

通常采用 `cpu 核数 + 1` 能够实现最优的 CPU 利用率，+1 是保证当线程由于页缺失故障（操作系统）或其它原因导致暂停时，额外的这个线程就能顶上去，保证 CPU 时钟周期不被浪费

#### 3.2 I/O 密集型运算

CPU 不总是处于繁忙状态，例如，当你执行业务计算时，这时候会使用 CPU 资源，但当你执行 I/O 操作时、远程 RPC 调用时，包括进行数据库操作时，这时候 CPU 就闲下来了，你可以利用多线程提高它的利用率。

经验公式如下

线程数 = 核数 \* 期望 CPU 利用率 \* 总时间(CPU计算时间+等待时间) / CPU 计算时间

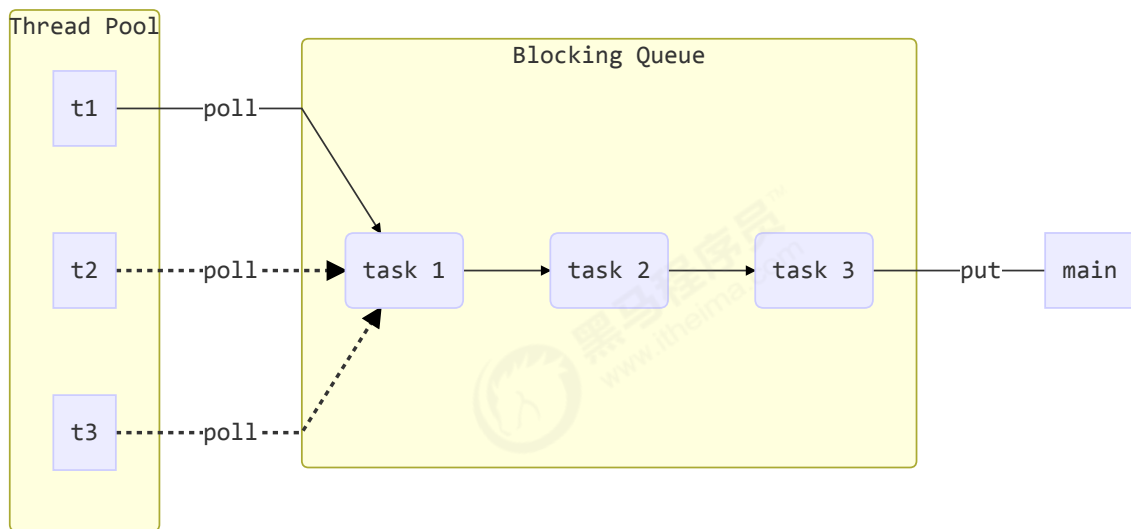
例如 4 核 CPU 计算时间是 50%，其它等待时间是 50%，期望 cpu 被 100% 利用，套用公式

$4 * 100\% * 100\% / 50\% = 8$

例如 4 核 CPU 计算时间是 10%，其它等待时间是 90%，期望 cpu 被 100% 利用，套用公式

$4 * 100\% * 100\% / 10\% = 40$

### 4. 自定义线程池



#### 步骤1：自定义拒绝策略接口

```
@FunctionalInterface // 拒绝策略
interface RejectPolicy<T> {
    void reject(BlockingQueue<T> queue, T task);
}
```

#### 步骤2：自定义任务队列

```
class BlockingQueue<T> {
    // 1. 任务队列
    private Deque<T> queue = new ArrayDeque<>();

    // 2. 锁
    private ReentrantLock lock = new ReentrantLock();

    // 3. 生产者条件变量
    private Condition fullWaitSet = lock.newCondition();

    // 4. 消费者条件变量
    private Condition emptyWaitSet = lock.newCondition();

    // 5. 容量
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }
}
```



```
// 带超时阻塞获取
public T poll(long timeout, TimeUnit unit) {
    lock.lock();
    try {
        // 将 timeout 统一转换为 纳秒
        long nanos = unit.toNanos(timeout);
        while (queue.isEmpty()) {
            try {
                // 返回值是剩余时间
                if (nanos <= 0) {
                    return null;
                }
                nanos = emptyWaitSet.awaitNanos(nanos);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        T t = queue.removeFirst();
        fullWaitSet.signal();
        return t;
    } finally {
        lock.unlock();
    }
}

// 阻塞获取
public T take() {
    lock.lock();
    try {
        while (queue.isEmpty()) {
            try {
                emptyWaitSet.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        T t = queue.removeFirst();
        fullWaitSet.signal();
        return t;
    } finally {
        lock.unlock();
    }
}

// 阻塞添加
public void put(T task) {
    lock.lock();
    try {
        while (queue.size() == capacity) {
            try {
                log.debug("等待加入任务队列 {} ...", task);
                fullWaitSet.await();
            }
        }
    }
}
```



```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    log.debug("加入任务队列 {}", task);
    queue.addLast(task);
    emptyWaitSet.signal();
} finally {
    lock.unlock();
}
}

// 带超时时间阻塞添加
public boolean offer(T task, long timeout, TimeUnit timeUnit) {
    lock.lock();
    try {
        long nanos = timeUnit.toNanos(timeout);
        while (queue.size() == capacity) {
            try {
                if(nanos <= 0) {
                    return false;
                }
                log.debug("等待加入任务队列 {} ...", task);
                nanos = fullWaitSet.awaitNanos(nanos);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        log.debug("加入任务队列 {}", task);
        queue.addLast(task);
        emptyWaitSet.signal();
        return true;
    } finally {
        lock.unlock();
    }
}

public int size() {
    lock.lock();
    try {
        return queue.size();
    } finally {
        lock.unlock();
    }
}

public void tryPut(RejectPolicy<T> rejectPolicy, T task) {
    lock.lock();
    try {
        // 判断队列是否满
        if(queue.size() == capacity) {
            rejectPolicy.reject(this, task);
        } else { // 有空闲
```



```
        log.debug("加入任务队列 {}", task);
        queue.addLast(task);
        emptyWaitSet.signal();
    }
} finally {
    lock.unlock();
}
}
}
```

### 步骤3：自定义线程池

```
class ThreadPool {
    // 任务队列
    private BlockingQueue<Runnable> taskQueue;

    // 线程集合
    private HashSet<Worker> workers = new HashSet<>();

    // 核心线程数
    private int coreSize;

    // 获取任务时的超时时间
    private long timeout;

    private TimeUnit timeUnit;

    private RejectPolicy<Runnable> rejectPolicy;

    // 执行任务
    public void execute(Runnable task) {
        // 当任务数没有超过 coreSize 时，直接交给 worker 对象执行
        // 如果任务数超过 coreSize 时，加入任务队列暂存
        synchronized (workers) {
            if(workers.size() < coreSize) {
                Worker worker = new Worker(task);
                log.debug("新增 worker{}", {}, worker, task);
                workers.add(worker);
                worker.start();
            } else {
                taskQueue.put(task);
                // 1) 死等
                // 2) 带超时等待
                // 3) 让调用者放弃任务执行
                // 4) 让调用者抛出异常
                // 5) 让调用者自己执行任务
                taskQueue.tryPut(rejectPolicy, task);
            }
        }
    }

    public ThreadPool(int coreSize, long timeout, TimeUnit timeUnit, int queueCapacity,
```

```
RejectPolicy<Runnable> rejectPolicy) {
    this.coreSize = coreSize;
    this.timeout = timeout;
    this.timeUnit = timeUnit;
    this.taskQueue = new BlockingQueue<>(queueCapacity);
    this.rejectPolicy = rejectPolicy;
}

class Worker extends Thread{
    private Runnable task;

    public Worker(Runnable task) {
        this.task = task;
    }

    @Override
    public void run() {
        // 执行任务
        // 1) 当 task 不为空，执行任务
        // 2) 当 task 执行完毕，再接着从任务队列获取任务并执行
        while(task != null || (task = taskQueue.take()) != null) {
            while(task != null || (task = taskQueue.poll(timeout, timeUnit)) != null) {
                try {
                    log.debug("正在执行...{}", task);
                    task.run();
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    task = null;
                }
            }
            synchronized (workers) {
                log.debug("worker 被移除{}", this);
                workers.remove(this);
            }
        }
    }
}
```

#### 步骤4：测试

```
public static void main(String[] args) {
    ThreadPool threadPool = new ThreadPool(1,
        1000, TimeUnit.MILLISECONDS, 1, (queue, task)->{
        // 1. 死等
        queue.put(task);
        // 2) 带超时等待
        queue.offer(task, 1500, TimeUnit.MILLISECONDS);
        // 3) 让调用者放弃任务执行
        log.debug("放弃{}", task);
        // 4) 让调用者抛出异常

        throw new RuntimeException("任务执行失败 " + task);
    });
}
```



```
// 5) 让调用者自己执行任务
task.run();
});
for (int i = 0; i < 4; i++) {
    int j = i;
    threadPool.execute(() -> {
        try {
            Thread.sleep(1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        log.debug("{} ", j);
    });
}
```

## 终止模式之两阶段终止模式

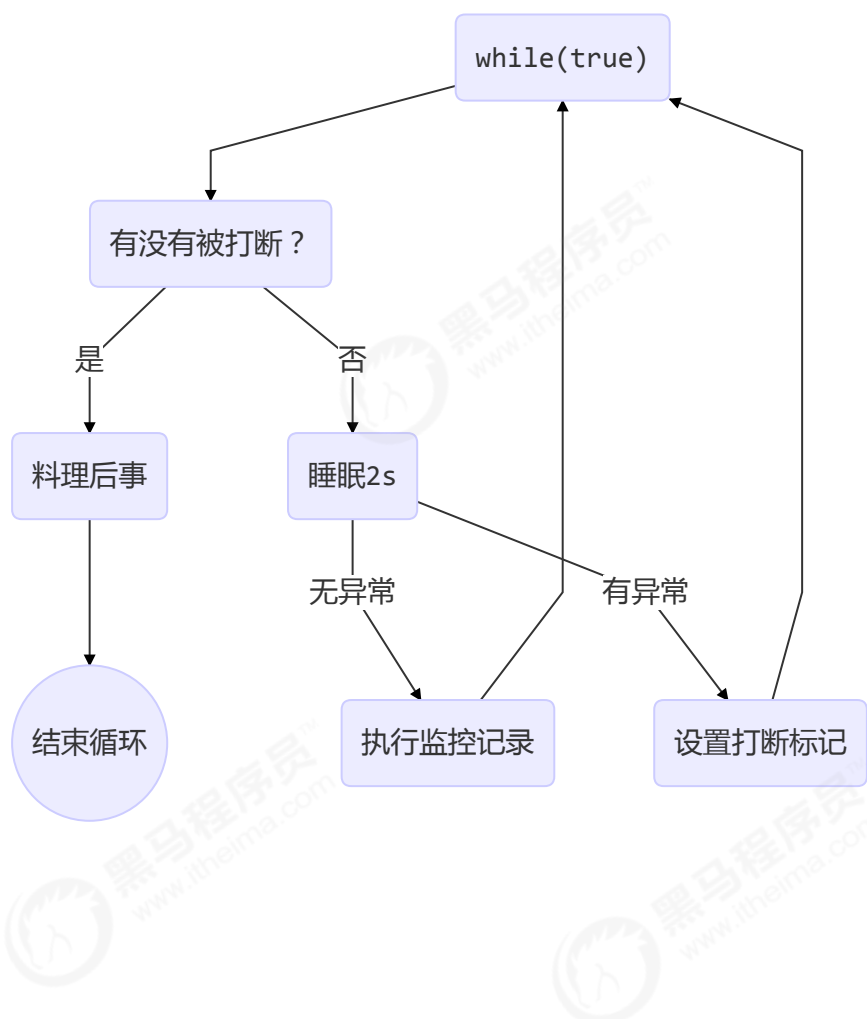
Two Phase Termination

在一个线程 T1 中如何“优雅”终止线程 T2？这里的【优雅】指的是给 T2 一个料理后事的机会。

### 1. 错误思路

- 使用线程对象的 stop() 方法停止线程
  - stop 方法会真正杀死线程，如果这时线程锁住了共享资源，那么当它被杀死后就再也没有机会释放锁，其它线程将永远无法获取锁
- 使用 System.exit(int) 方法停止线程
  - 目的仅是停止一个线程，但这种做法会让整个程序都停止

### 2. 两阶段终止模式



## 2.1 利用 isInterrupted

`interrupt` 可以打断正在执行的线程，无论这个线程是在 `sleep`，`wait`，还是正常运行

```
class TPTInterrupt {
    private Thread thread;

    public void start(){
        thread = new Thread(() -> {
            while(true) {
                Thread current = Thread.currentThread();
                if(current.isInterrupted()) {
                    log.debug("料理后事");
                    break;
                }
                try {
                    Thread.sleep(1000);
                    log.debug("将结果保存");
                } catch (InterruptedException e) {
                    current.interrupt();
                }
            }
        });
    }
}
```

```
    }  
    // 执行监控操作  
    }  
    }, "监控线程");  
    thread.start();  
}  
  
public void stop() {  
    thread.interrupt();  
}  
}
```

调用

```
TPTInterrupt t = new TPTInterrupt();  
t.start();  
  
Thread.sleep(3500);  
log.debug("stop");  
t.stop();
```

结果

```
11:49:42.915 c.TwoPhaseTermination [监控线程] - 将结果保存  
11:49:43.919 c.TwoPhaseTermination [监控线程] - 将结果保存  
11:49:44.919 c.TwoPhaseTermination [监控线程] - 将结果保存  
11:49:45.413 c.TestTwoPhaseTermination [main] - stop  
11:49:45.413 c.TwoPhaseTermination [监控线程] - 料理后事
```

## 2.2 利用停止标记

```
// 停止标记用 volatile 是为了保证该变量在多个线程之间的可见性  
// 我们的例子中，即主线程把它修改为 true 对 t1 线程可见  
class TPTVolatile {  
    private Thread thread;  
    private volatile boolean stop = false;  
  
    public void start(){  
        thread = new Thread(() -> {  
            while(true) {  
                Thread current = Thread.currentThread();  
                if(stop) {  
                    log.debug("料理后事");  
                    break;  
                }  
                try {  
                    Thread.sleep(1000);  
                    log.debug("将结果保存");  
                } catch (InterruptedException e) {
```

```
        }  
        // 执行监控操作  
    }  
    }, "监控线程");  
    thread.start();  
}  
  
public void stop() {  
    stop = true;  
    thread.interrupt();  
}  
}
```

调用

```
TPTVolatile t = new TPTVolatile();  
t.start();  
  
Thread.sleep(3500);  
log.debug("stop");  
t.stop();
```

结果

```
11:54:52.003 c.TPTVolatile [监控线程] - 将结果保存  
11:54:53.006 c.TPTVolatile [监控线程] - 将结果保存  
11:54:54.007 c.TPTVolatile [监控线程] - 将结果保存  
11:54:54.502 c.TestTwoPhaseTermination [main] - stop  
11:54:54.502 c.TPTVolatile [监控线程] - 料理后事
```

## 案例：JVM 内存监控

## 线程安全单例

单例模式有很多实现方法，饿汉、懒汉、静态内部类、枚举类，试分析每种实现下获取单例对象（即调用 getInstance）时的线程安全，并思考注释中的问题

饿汉式：类加载就会导致该单实例对象被创建

懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建

### 1. 饿汉单例

```
// 问题1：为什么加 final  
// 问题2：如果实现了序列化接口，还要做什么来防止反序列化破坏单例  
public final class Singleton implements Serializable {  
    // 问题3：为什么设置为私有？是否能防止反射创建新的实例？
```



```
private Singleton() {}  
// 问题4：这样初始化是否能保证单例对象创建时的线程安全？  
private static final Singleton INSTANCE = new Singleton();  
// 问题5：为什么提供静态方法而不是直接将 INSTANCE 设置为 public，说出你知道的理由  
public static Singleton getInstance() {  
    return INSTANCE;  
}  
public Object readResolve() {  
    return INSTANCE;  
}  
}
```

## 2. 枚举单例

```
// 问题1：枚举单例是如何限制实例个数的  
// 问题2：枚举单例在创建时是否有并发问题  
// 问题3：枚举单例能否被反射破坏单例  
// 问题4：枚举单例能否被反序列化破坏单例  
// 问题5：枚举单例属于懒汉式还是饿汉式  
// 问题6：枚举单例如果希望加入一些单例创建时的初始化逻辑该如何做  
enum Singleton {  
    INSTANCE;  
}
```

## 3. 懒汉单例

```
public final class Singleton {  
    private Singleton() { }  
    private static Singleton INSTANCE = null;  
    // 分析这里的线程安全，并说明有什么缺点  
    public static synchronized Singleton getInstance() {  
        if( INSTANCE != null ){  
            return INSTANCE;  
        }  
        INSTANCE = new Singleton();  
        return INSTANCE;  
    }  
}
```

## 4. DCL 懒汉单例

```
public final class Singleton {  
    private Singleton() { }  
    // 问题1：解释为什么要加 volatile ?  
    private static volatile Singleton INSTANCE = null;  
  
    // 问题2：对比实现3，说出这样做的意义  
    public static Singleton getInstance() {  
        if (INSTANCE != null) {
```

```
        return INSTANCE;
    }
    synchronized (Singleton.class) {
        // 问题3：为什么还要在这里加为空判断，之前不是判断过了吗
        if (INSTANCE != null) {           // t2
            return INSTANCE;
        }
        INSTANCE = new Singleton();
        return INSTANCE;
    }
}
```

## 5. 静态内部类懒汉单例

```
public final class Singleton {
    private Singleton() { }
    // 问题1：属于懒汉式还是饿汉式
    private static class LazyHolder {
        static final Singleton INSTANCE = new Singleton();
    }
    // 问题2：在创建时是否有并发问题
    public static Singleton getInstance() {
        return LazyHolder.INSTANCE;
    }
}
```

## 享元模式

### 1. 简介

**定义** 英文名称：Flyweight pattern. 当需要重用数量有限的同一类对象时

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

**出自** "Gang of Four" design patterns

**归类** Structural patterns

### 2. 体现

#### 2.1 包装类

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，在这个范围之内会重用对象，大于这个范围，才会新建 Long 对象：



```
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);  
}
```

#### 注意：

- 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

## 2.2 String 串池

## 2.3 BigDecimal BigInteger

## 3. DIY

例如：一个线上商城应用，QPS 达到数千，如果每次都重新创建和关闭数据库连接，性能会受到极大影响。这时预先创建好一批连接，放入连接池。一次请求到达后，从连接池获取连接，使用完毕后再还回连接池，这样既节约了连接的创建和关闭时间，也实现了连接的重用，不至于让庞大的连接数压垮数据库。

```
class Pool {  
    // 1. 连接池大小  
    private final int poolSize;  
  
    // 2. 连接对象数组  
    private Connection[] connections;  
  
    // 3. 连接状态数组 0 表示空闲，1 表示繁忙  
    private AtomicIntegerArray states;  
  
    // 4. 构造方法初始化  
    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("连接" + (i+1));  
        }  
    }  
  
    // 5. 借连接  
    public Connection borrow() {
```



```
while(true) {
    for (int i = 0; i < poolSize; i++) {
        // 获取空闲连接
        if(states.get(i) == 0) {
            if (states.compareAndSet(i, 0, 1)) {
                log.debug("borrow {}", connections[i]);
                return connections[i];
            }
        }
    }
    // 如果没有空闲连接，当前线程进入等待
    synchronized (this) {
        try {
            log.debug("wait...");
            this.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// 6. 归还连接
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;
        }
    }
}

class MockConnection implements Connection {
    // 实现略
}
```

使用连接池：



```
Pool pool = new Pool(2);
for (int i = 0; i < 5; i++) {
    new Thread(() -> {
        Connection conn = pool.borrow();
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        pool.free(conn);
    }).start();
}
```

以上实现没有考虑：

- 连接的动态增长与收缩
- 连接保活（可用性检测）
- 等待超时处理
- 分布式 hash

对于关系型数据库，有比较成熟的连接池实现，例如c3p0, druid等 对于更通用的对象池，可以考虑使用apache commons pool，例如redis连接池可以参考jedis中关于连接池的实现