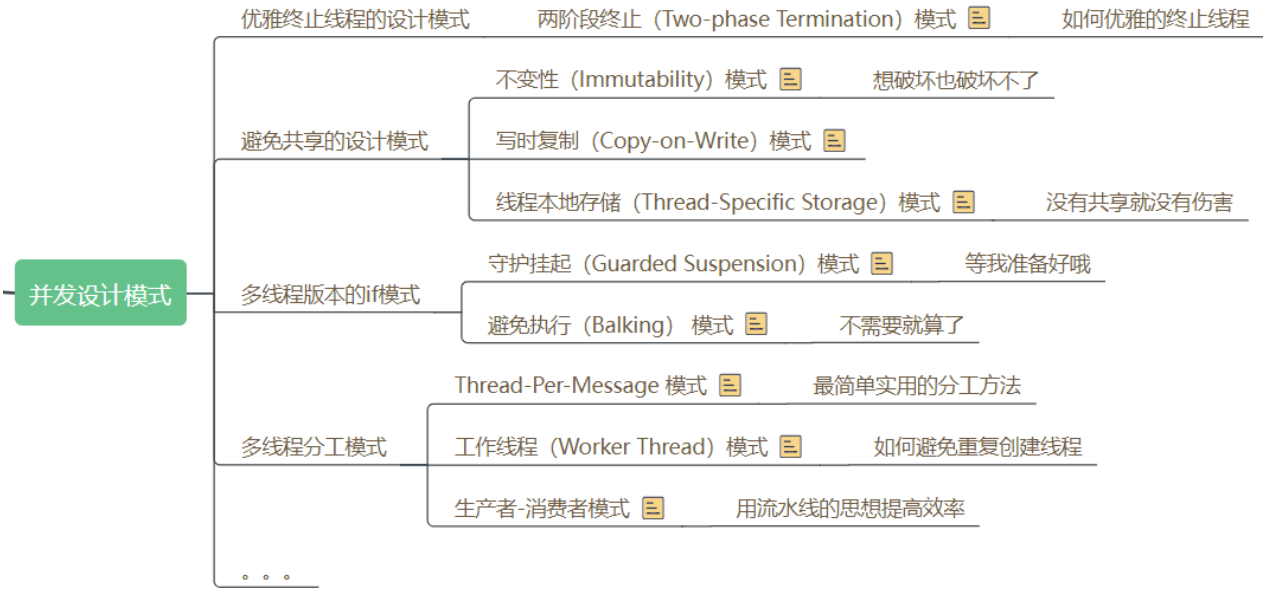
主讲老师： Fox

> 有道笔记地址：https://note.youdao.com/s/K8sOCHWU

常用并发设计模式

https://www.processon.com/view/link/615d4a610e3e74663e97fa0e



1. 优雅终止线程的设计模式

思考：在一个线程 T1 中如何优雅的终止线程 T2?

正确思路：两阶段终止模式

1.1 两阶段终止（Two-phase Termination）模式——优雅的终止线程

两阶段终止（Two-phase Termination）模式是一种用于优雅终止线程的设计模式。该模式的基本思想是通过两个阶段来终止线程，**第一个阶段是发送终止请求，第二个阶段是等待线程终止。**

思考：第一阶段，在Java中如何发送终止请求?

回顾一下Java线程的生命周期：

Java 线程进入终止状态的前提是线程进入 RUNNABLE 状态，而实际上线程也可能处在休眠状态，也就是说，我们要想终止一个线程，首先要把线程的状态从休眠状态转换到 RUNNABLE 状态。

利用java线程中断机制的interrupt() 方法，可以让线程从休眠状态转换到RUNNABLE 状态。

思考：第二阶段，线程转换到 RUNNABLE 状态之后，我们如何再将其终止呢?

RUNNABLE 状态转换到终止状态，优雅的方式是让 Java 线程自己执行完 run() 方法，所以一般我们采用的方法是设置一个标志位，然后线程会在合适的时机检查这个标志位，如果发现符合终止条件，则自动退出 run() 方法。

综合上面这两点，我们能总结出终止指令，其实包括两方面内容：**interrupt() 方法和线程终止的标志位。**

两阶段终止模式可以带来多种好处，例如：

1. 优雅终止：两阶段终止模式可以优雅地终止线程，避免突然终止线程带来的副作用。
2. 安全性：两阶段终止模式可以在线程终止前执行必要的清理工作，以确保程序的安全性和稳定性。
3. 灵活性：两阶段终止模式可以根据具体情况灵活地设置终止条件和清理工作。

1.2 使用场景

两阶段终止模式适用于需要优雅终止线程的场景，例如：

1. 服务器应用程序：在服务器应用程序中，需要处理大量的请求和数据，并且需要在终止时正确地保存和释放资源，以避免数据丢失和资源泄漏。
2. 大规模并发系统：在大规模并发系统中，线程数量可能非常多，并且需要在终止时正确地关闭和释放所有的线程和资源。
3. 定时任务系统：在定时任务系统中，需要在任务执行完毕后正确地终止任务线程，并清理相关资源。
4. 数据处理系统：在数据处理系统中，需要在处理完所有数据后正确地终止线程，并清理相关资源。
5. 消息订阅系统：在消息订阅系统中，需要在订阅结束后正确地终止订阅线程，并清理相关资源。

总之，两阶段终止模式适用于需要优雅终止线程的各种场景，可以提高程序的可靠性和可维护性。在应用该模式时，需要注意正确设计终止条件和清理工作，以避免出现线程安全问题和资源泄漏问题。

利用两阶段终止模式设计优雅终止监控操作

在多线程程序中，如果有一些线程需要执行长时间的监控或者轮询操作，可以使用两阶段终止模式来终止这些线程的执行。使用两阶段终止模式可以保证监控线程在执行终止操作时能够安全地释放资源和退出线程。同时，该模式还可以保证监控线程在终止前能够完成必要的清理工作，从而避免资源泄露和其他问题。

```java
public class MonitorThread extends Thread {
    //在监控线程中添加一个volatile类型的标志变量，用于标识是否需要终止线程的执行
    private volatile boolean terminated = false;

    public void run() {
        while (!terminated) {
            // 执行监控操作
            System.out.println("监控线程正在执行监控操作...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 执行清理操作
        System.out.println("监控线程正在执行清理操作...");
        releaseResources();
    }

    public void terminate() {
        //设置标志变量为true，并等待一段时间
        terminated = true;
        try {
            join(5000); // 等待5秒钟,期间监控线程会检查terminated的状态
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void releaseResources() {
        // 释放资源和进行必要的清理工作
        System.out.println("监控线程正在释放资源和进行必要的清理工作...");
    }

    public static void main(String[] args) throws InterruptedException {
        MonitorThread thread = new MonitorThread();
        //启动监控线程
        thread.start();
```

```
39          //主线程休眠期间，监控线程在执行监控操作
40          Thread.sleep(10000);
41          //终止监控线程
42          thread.terminate();
43
44
45          Thread.sleep(100000);
46      }
47  }
```

在使用两阶段终止模式终止线程的同时，还可以结合中断机制实现更加可靠和灵活的线程终止操作。使用中断机制可以让线程在终止前及时响应中断请求，并退出线程的阻塞状态。同时，还可以通过检查中断状态和标志变量的状态来判断是否需要终止线程的执行，从而实现更加可靠和灵活的线程终止操作。

```java
public class MonitorThread2 extends Thread {
    //在监控线程中添加一个volatile类型的标志变量，用于标识是否需要终止线程的执行
    private volatile boolean terminated = false;

    public void run() {
        while (!Thread.interrupted()&&!terminated) {
            // 执行监控操作
            System.out.println("监控线程正在执行监控操作...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("监控线程被中断，准备退出...");
                Thread.currentThread().interrupt();
                e.printStackTrace();
            }
        }
        // 执行清理操作
        System.out.println("监控线程正在执行清理操作...");
        releaseResources();
    }

    public void terminate() {
        //设置标志变量为true，并等待一段时间
        terminated = true;
        try {
            join(5000); // 等待5秒钟,期间监控线程会检查terminated的状态
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void releaseResources() {
        // 释放资源和进行必要的清理工作
        System.out.println("监控线程正在释放资源和进行必要的清理工作...");
    }

    public static void main(String[] args) throws InterruptedException {
        MonitorThread2 thread = new MonitorThread2();
```

```
39          //启动监控线程
40          thread.start();
41          //主线程休眠期间，监控线程在执行监控操作
42          Thread.sleep(10000);
43          //为监控线程设置中断标志位
44          thread.interrupt();
45          //终止监控线程
46          //thread.terminate();
47
48
49          Thread.sleep(100000);
50      }
51  }
```

如何优雅地终止线程池

Java 领域用的最多的还是线程池，而不是手动地创建线程。那我们该如何优雅地终止线程池呢？

线程池提供了两个终止线程池的方法：

- shutdown()方法会停止线程池接受新的任务，并等待线程池中的所有任务执行完毕，然后关闭线程池。在调用shutdown()方法后，线程池不再接受新的任务，但是会将任务队列中的任务继续执行直到队列为空。如果线程池中的任务正在执行，但是还没有执行完毕，线程池会等待所有任务执行完毕后再关闭线程池。
- shutdownNow()方法会停止线程池接受新的任务，并尝试中断正在执行任务的线程，然后关闭线程池。在调用shutdownNow()方法后，线程池不再接受新的任务，同时会中断正在执行任务的线程并返回一个未执行的任务列表。该方法会调用每个任务的interrupt()方法尝试中断任务执行的线程，但是并不能保证线程一定会被中断，因为线程可以选择忽略中断请求。

```java
public class ThreadPoolDemo {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++) {
            executorService.submit(() -> {
                try {
                    // 执行任务操作
                    System.out.println(Thread.currentThread().getName() + "正在执行任务...");
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    // 重新设置中断状态
                    Thread.currentThread().interrupt();
                    e.printStackTrace();
                } finally {
                    System.out.println(Thread.currentThread().getName() + "任务执行完毕");
                }
            });
        }

        // 停止线程池接受新的任务，但不能强制停止已经提交的任务
        executorService.shutdown();

        // 等待线程池中的任务执行完毕，或者超时时间到达
        boolean terminated = executorService.awaitTermination(8, TimeUnit.SECONDS);
        if (!terminated) {
            // 如果线程池中还有未执行完毕的任务，则调用线程池的shutdownNow方法，中断所有正在执行任务的线程
            // 如果有还没开始执行的任务，则返回未执行的任务列表
            List<Runnable> tasks = executorService.shutdownNow();
            System.out.println("剩余未执行的任务数: " + tasks.size());
        }

    }
}
```

运行该程序，可以看到线程池会依次执行10个任务，然后优雅地终止线程池的执行，中断所有正在执行的任务。

1.3 注意事项

两阶段终止模式是一种应用很广泛的并发设计模式，在 Java 语言中使用两阶段终止模式来优雅地终止线程，需要注意两个关键点：

- 一个是仅检查终止标志位是不够的，因为线程的状态可能处于休眠态；
- 另一个是仅检查线程的中断状态也是不够的，因为我们依赖的第三方类库很可能没有正确处理中断异常，例如第三方类库在捕获到 Thread.sleep() 方法抛出的中断异常后，没有重新设置线程的中断状态，那么就会导致线程不能够正常终止。所以我们可以自定义线程的终止标志位用于终止线程。

当你使用 Java 的线程池来管理线程的时候，需要依赖线程池提供的 shutdown() 和 shutdownNow() 方法来终止线程池。不过在使用时需要注意它们的应用场景，尤其是在使用 shutdownNow() 的时候，一定要谨慎。

2. 避免共享的设计模式

不变性（Immutability）模式，写时复制（Copy-on-Write）模式，线程本地存储（Thread-Specific Storage）模式本质上都是为了避免共享。

- 使用时需要注意不变性模式的属性的不可变性
- 写时复制模式需要注意拷贝的性能问题
- 线程本地存储模式需要注意异步执行问题。

2.1 不变性（Immutability）模式——想破坏也破坏不了

“多个线程同时读写同一共享变量存在并发问题”，这里的必要条件之一是读写，如果只有读，而没有写，是没有并发问题的。解决并发问题，其实最简单的办法就是让共享变量只有读操作，而没有写操作。这个办法如此重要，以至于被上升到了一种解决并发问题的设计模式：不变性（Immutability）模式。

不变性模式是一种创建不可变对象的设计模式，即对象一旦创建后，就不能再进行修改。**在多线程环境下，使用不可变对象可以避免线程安全问题，并提高程序的性能和可读性。**

使用不变性模式可以带来多种好处，例如：

1. 线程安全性：不可变对象在多线程环境下不需要进行同步操作，可以避免线程安全问题。
2. 可读性：不可变对象的状态在创建后不可修改，可以更加清晰地表达对象的含义和作用。
3. 性能：由于不可变对象的状态不可变，可以进行更加有效的缓存和优化操作。
4. 可测试性：不可变对象对单元测试非常友好，可以更容易地进行测试和验证。

不变性模式虽然有很多优点，但也有一些限制。例如，不可变对象的状态一旦创建后就无法修改，需要重新创建一个新的对象。因此，在需要频繁修改对象状态的场景下，不可变对象可能不太适用。同时，在不可变对象之间存在引用关系的情况下，需要注意对象状态的变化。

使用场景

不变性模式适用于需要确保对象状态不变的场景，例如：

1. 缓存：在缓存系统中，需要缓存一些数据，以避免重复计算和频繁访问。由于缓存数据是共享的，为了避免数据被修改，通常使用不变性模式来确保缓存数据的不变性。
2. 值对象：在一些系统中，需要定义一些值对象来表示一些常量或者不可变的对象。使用不变性模式可以确保这些值对象的状态不变，从而避免出现错误和不一致。
3. 配置信息：在一些系统中，需要读取一些配置信息来配置系统参数和行为。由于配置信息通常是不变的，可以使用不变性模式来确保配置信息的不变性。

如何实现不变性模式

不变性模式的主要思想是通过**将对象的状态设置为final和私有，并提供只读方法来保证对象的不可变性。**在创建不可变对象时，需要确保对象的所有属性都是不可变的，即在创建后不会被修改。同时，还需要注意不可变对象之间的引用关系，以避免出现对象的状态变化。

jdk中很多类都具备不可变性，例如经常用到的 String 和 Long、Integer、Double 等基础类型的包装类都具备不可变性，这些对象的线程安全性都是靠不可变性来保证的。它们都严格遵守了不可变类的三点要求：类和属性都是 final 的，所有方法均是只读的。

使用 Immutability 模式的注意事项

在使用 Immutability 模式的时候，需要注意以下两点：

- 对象的所有属性都是 final 的，并不能保证不可变性；
- 不可变对象也需要正确发布。

在 Java 语言中，final 修饰的属性一旦被赋值，就不可以再修改，但是如果属性的类型是普通对象，那么这个普通对象的属性是可以被修改的。所以，在使用 Immutability 模式的时候一定要确认保持不变性的边界在哪里，是否要求属性对象也具备不可变性。

下面的代码中，Bar 的属性 foo 虽然是 final 的，依然可以通过 setAge() 方法来设置 foo 的属性 age。

```
class Foo{
  int age=0;
  int name="abc";
}
final class Bar {
  final Foo foo;
  void setAge(int a){
    foo.age=a;
  }
}
```

可变对象虽然是线程安全的，但是并不意味着引用这些不可变对象的对象就是线程安全的。

下面的代码中，Foo 具备不可变性，线程安全，但是类 Bar 并不是线程安全的，类 Bar 中持有对 Foo 的引用 foo，对 foo 这个引用的修改在多线程中并不能保证可见性和原子性。

```

//Foo线程安全
final class Foo{
  final int age=0;
  final String name="abc";
}
//Bar线程不安全
class Bar {
  Foo foo;
  void setFoo(Foo f){
    this.foo=f;
  }
}
```

2.2 写时复制（Copy-on-Write）模式

在多线程环境下，Copy-on-Write模式可以提高共享数据的并发性能。该模式的基本思想是在共享数据被修改时，先将数据复制一份，然后对副本进行修改，最后再将副本替换为原始的共享数据。通过这种方式，可以避免多个线程同时访问同一个共享数据造成的竞争和冲突。

不可变对象的写操作往往都是使用 Copy-on-Write 方法解决的，当然 Copy-on-Write 的应用领域并不局限于 Immutability 模式。

Copy-on-Write 才是最简单的并发解决方案，很多人都在无意中把它忽视了。它是如此简单，以至于 Java 中的基本数据类型 String、Integer、Long 等都是基于 Copy-on-Write 方案实现的。

Copy-on-Write 缺点就是消耗内存，每次修改都需要复制一个新的对象出来，好在随着自动垃圾回收（GC）算法的成熟以及硬件的发展，这种内存消耗已经渐渐可以接受了。所以在实际工作中，如果写操作非常少（读多写少的场景），可以尝试使用 Copy-on-Write。

使用场景

在Java中，**CopyOnWriteArrayList 和 CopyOnWriteArraySet** 这两个 Copy-on-Write 容器，它们背后的设计思想就是 Copy-on-Write；通过 Copy-on-Write 这两个容器实现的读操作是无锁的，由于无锁，所以将读操作的性能发挥到了极致。

Copy-on-Write 在操作系统领域也有广泛的应用。类 Unix 的操作系统中创建进程的 API 是 fork()，传统的 fork() 函数会创建父进程的一个完整副本，例如父进程的地址空间现在用到了 1G 的内存，那么 fork() 子进程的时候要复制父进程整个进程的地址空间（占有 1G 内存）给子进程，这个过程是很耗时的。而 Linux 中fork() 子进程的时候，并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间；只用在父进程或者子进程需要写入的时候才会复制地址空间，从而使父子进程拥有各自的地址空间。

Copy-on-Write 最大的应用领域还是在函数式编程领域。函数式编程的基础是不可变性（Immutability），所以函数式编程里面所有的修改操作都需要 Copy-on-Write 来解决。

像一些RPC框架还有服务注册中心，也会利用Copy-on-Write设计思想维护服务路由表。路由表是典型的读多写少，而且路由表对数据的一致性要求并不高，一个服务提供方从上线到反馈到客户端的路由表里，即便有 5 秒钟延迟，很多时候也都是能接受的。

2.3 线程本地存储（Thread-Specific Storage） 模式——没有共享就没有伤害

线程本地存储模式用于解决多线程环境下的数据共享和数据隔离问题。该模式的基本思想是为每个线程创建独立的存储空间，用于存储线程私有的数据。通过这种方式，可以保证线程之间的数据隔离和互不干扰。在 Java 标准类库中，ThreadLocal 类实现了该模式。

线程本地存储模式本质上是一种避免共享的方案，由于没有共享，所以自然也就没有并发问题。如果你需要在并发场景中使用一个线程不安全的工具类，最简单的方案就是避免共享。避免共享有两种方案，一种方案是将这个工具类作为局部变量使用，另外一种方案就是线程本地存储模式。这两种方案，局部变量方案的缺点是在高并发场景下会频繁创建对象，而线程本地存储方案，每个线程只需要创建一个工具类的实例，所以不存在频繁创建对象的问题。

使用场景

线程本地存储模式通常用于以下场景：

1. **保存上下文信息**：在多线程环境中，每个线程都有自己的执行上下文，包括线程的状态、环境变量、运行时状态等。线程本地存储可以用来保存这些上下文信息，使得每个线程都可以独立地访问和修改自己的上下文信息。
2. **管理线程安全对象**：在多线程环境中，共享对象通常需要进行同步操作以避免竞争条件。但是，有些对象是线程安全的，可以被多个线程同时访问而不需要同步操作。线程本地存储可以用来管理这些线程安全对象，使得每个线程都可以独立地访问自己的对象实例，而不需要进行同步操作。
3. **实现线程特定的行为**：有些应用程序需要在每个线程中执行特定的行为，例如跟踪日志、统计数据、授权访问等。线程本地存储可以用来实现这些线程特定的行为，使得每个线程都可以独立地执行自己的行为逻辑，而不需要与其他线程进行协调。

需要注意的是，线程本地存储虽然可以提高性能，但也可能会导致内存泄漏和数据一致性问题。因此，在使用线程本地存储时，需要仔细考虑数据的生命周期和线程的使用情况，避免出现潜在的问题。

注意：在线程池中使用ThreadLocal 需要避免内存泄漏和线程安全的问题

```
ExecutorService es;
ThreadLocal tl;
es.execute(()->{
  //ThreadLocal增加变量
  tl.set(obj);
  try {
    // 省略业务逻辑代码
  }finally {
    //手动清理ThreadLocal
    tl.remove();
  }
});
```

3. 多线程版本的if模式

守护挂起（Guarded Suspension）模式和避免执行（Balking）模式属于多线程版本的if模式

- 守护挂起模式需要注意性能。
- 避免重复执行模式需要注意竞态问题。

3.1 守护挂起（Guarded Suspension）模式——等我准备好哦

守护挂起模式是通过让线程等待来保护实例的安全性，即守护-挂起模式。在多线程开发中，常常为了提高应用程序的并发性，会将一个任务分解为多个子任务交给多个线程并行执行，而多个线程之间相互协作时，仍然会存在一个线程需要等待另外的线程完成后继续下一步操作。而Guarded Suspension模式可以帮助我们解决上述的等待问题。

守护挂起模式允许多个线程对实例资源进行访问，但是实例资源需要对资源的分配做出管理。

守护挂起模式也常被称作 Guarded Wait 模式、Spin Lock 模式（因为使用了 while 循环去等待），它还有一个更形象的非官方名字：多线程版本的 if。

- 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
- 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
- 等待唤醒机制的规范实现。此模式依赖于Java线程的等待唤醒机制：
    - synchronized+wait/notify/notifyAll
    - reentrantLock+Condition(await/singal/singalAll)
    - cas+park/unpark

> 等待唤醒机制底层原理： linux pthread_mutex_lock/unlock pthread_cond_wait/singal
> 解决线程之间的协作不可避免会用到等待唤醒机制

使用场景

- 多线程环境下多个线程访问相同实例资源，从实例资源中获得资源并处理；
- 实例资源需要管理自身拥有的资源，并对请求线程的请求作出允许与否的判断；

守护挂起模式的实现

```java
public class GuardedObject<T> {
    //结果
    private T obj;
    //获取结果
    public T get(){
        synchronized (this){
            //没有结果等待    防止虚假唤醒
            while (obj==null){
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            return obj;
        }
    }
    //产生结果
    public void complete(T obj){
        synchronized (this){
            //获取到结果，给obj赋值
            this.obj = obj;
            //唤醒等待结果的线程
            this.notifyAll();
        }
    }
}
```

3.2 避免执行（Balking）模式——不需要就算了

Balking是“退缩不前”的意思。如果现在不适合执行这个操作，或者没必要执行这个操作，就停止处理，直接返回。当流程的执行顺序依赖于某个共享变量的场景，可以归纳为**多线程if模式**。Balking 模式常用于一个线程发现另一个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。

Balking模式是一种多个线程执行同一操作A时可以考虑的模式；在某一个线程B被阻塞或者执行其他操作时，其他线程同样可以完成操作A，而当线程B恢复执行或者要执行操作A时，因A已被执行，而无需线程B再执行，从而提高了B的执行效率。

Balking模式和Guarded Suspension模式一样，存在守护条件，如果守护条件不满足，则中断处理；这与Guarded Suspension模式不同，Guarded Suspension模式在守护条件不满足的时候会一直等待至可以运行。

使用场景

- sychronized轻量级锁膨胀逻辑， 只需要一个线程膨胀获取monitor对象
- DCL单例实现
- 服务组件的初始化

如何实现Balking模式

- 锁机制 （synchronized reentrantLock）
- cas
- 对于共享变量不要求原子性的场景，可以使用volatile

需要快速放弃的一个最常见的场景是各种编辑器提供的自动保存功能。自动保存功能的实现逻辑一般都是隔一定时间自动执行存盘操作，存盘操作的前提是文件做过修改，如果文件没有执行过修改操作，就需要快速放弃存盘操作。

```
boolean changed=false;
// 自动存盘操作
void autoSave(){
  synchronized(this){
    if (!changed) {
      return;
    }
    changed = false;
  }
  // 执行存盘操作
  // 省略且实现
  this.execSave();
}
// 编辑操作
void edit(){
  // 省略编辑逻辑
  ......
  change();
}
// 改变状态
void change(){
  synchronized(this){
    changed = true;
  }
}
```

Balking 模式有一个非常典型的应用场景就是单次初始化。

```
boolean inited = false;
synchronized void init(){
    if(inited){
      return;
    }
    //省略doInit的实现
    doInit();
    inited=true;
}
```

4. 多线程分工模式

Thread-Per-Message 模式、Worker Thread 模式和生产者 - 消费者模式属于多线程分工模式。

- Thread-Per-Message 模式需要注意线程的创建，销毁以及是否会导致OOM。
- Worker Thread 模式需要注意死锁问题，提交的任务之间不要有依赖性。
- 生产者 - 消费者模式可以直接使用线程池来实现

4.1 Thread-Per-Message 模式——最简单实用的分工方法

Thread-Per-Message 模式就是为每个任务分配一个独立的线程，这是一种最简单的分工方法。

应用场景

Thread-Per-Message 模式的一个最经典的应用场景是**网络编程里服务端的实现**，服务端为每个客户端请求创建一个独立的线程，当线程处理完请求后，自动销毁，这是一种最简单的并发处理网络请求的方法。

```
final ServerSocketChannel  ssc=
ServerSocketChannel.open().bind(new InetSocketAddress(8080));
//处理请求
try {
  while (true) {
    // 接收请求
    SocketChannel sc = ssc.accept();
    // 每个请求都创建一个线程
    new Thread(()->{
      try {
        // 读Socket
        ByteBuffer rb = ByteBuffer.allocateDirect(1024);
        sc.read(rb);
        //模拟处理请求
        Thread.sleep(2000);
        // 写Socket
        ByteBuffer wb = (ByteBuffer)rb.flip();
        sc.write(wb);
        // 关闭Socket
        sc.close();
      }catch(Exception e){
        throw new UncheckedIOException(e);
      }
    }).start();
  }
} finally {
  ssc.close();
}

```

Thread-Per-Message 模式作为一种最简单的分工方案，Java 中使用会存在性能缺陷。在 Java 中的线程是一个重量级的对象，创建成本很高，一方面创建线程比较耗时，另一方面线程占用的内存也比较大。所以为每个请求创建一个新的线程并不适合高并发场景。为了解决这个缺点，Java 并发包里提供了线程池等工具类。

在其他编程语言里，例如 Go 语言，基于轻量级线程实现 Thread-Per-Message 模式就完全没有问题。

对于一些并发度没那么高的异步场景，例如定时任务，采用 Thread-Per-Message 模式是完全没有问题的。

4.2 Worker Thread模式——如何避免重复创建线程

要想有效避免线程的频繁创建、销毁以及 OOM 问题，就不得不提 Java 领域使用最多的 Worker Thread 模式。Worker Thread 模式可以类比现实世界里车间的工作模式：车间里的工人，有活儿了，大家一起干，没活儿了就聊聊天等着。Worker Thread 模式中 Worker Thread 对应到现实世界里，其实指的就是车间里的工人。

Worker Thread 模式实现

之前的服务端例子用线程池实现

```java
ExecutorService es = Executors.newFixedThreadPool(200);
final ServerSocketChannel ssc = ServerSocketChannel.open().bind(new
InetSocketAddress(8080));
//处理请求
try {
  while (true) {
    // 接收请求
    SocketChannel sc = ssc.accept();
    // 将请求处理任务提交给线程池
    es.execute(()->{
      try {
        // 读Socket
        ByteBuffer rb = ByteBuffer.allocateDirect(1024);
        sc.read(rb);
        //模拟处理请求
        Thread.sleep(2000);
        // 写Socket
        ByteBuffer wb =
          (ByteBuffer)rb.flip();
        sc.write(wb);
        // 关闭Socket
        sc.close();
      }catch(Exception e){
        throw new UncheckedIOException(e);
      }
    });
  }
} finally {
  ssc.close();
  es.shutdown();
}
```

应用场景

Worker Thread 模式能避免线程频繁创建、销毁的问题，而且能够限制线程的最大数量。Java 语言里可以直接使用线程池来实现 Worker Thread 模式，线程池是一个非常基础和优秀的工具类，甚至有些大厂的编码规范都不允许用 new Thread() 来创建线程，必须使用线程池。

4.3 生产者 - 消费者模式——用流水线的思想提高效率

Worker Thread 模式类比的是工厂里车间工人的工作模式。但其实在现实世界，工厂里还有一种流水线的工作模式，类比到编程领域，就是生产者 - 消费者模式。

生产者 - 消费者模式的核心是一个任务队列，生产者线程生产任务，并将任务添加到任务队列中，而消费者线程从任务队列中获取任务并执行。

```java
public class BlockingQueueExample {

    private static final int QUEUE_CAPACITY = 5;
    private static final int PRODUCER_DELAY_MS = 1000;
    private static final int CONSUMER_DELAY_MS = 2000;

    public static void main(String[] args) throws InterruptedException {
        // 创建一个容量为QUEUE_CAPACITY的阻塞队列
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(QUEUE_CAPACITY);

        // 创建一个生产者线程
        Runnable producer = () -> {
            while (true) {
                try {
                    // 在队列满时阻塞
                    queue.put("producer");

                    System.out.println("生产了一个元素，队列中元素个数：" + queue.size());
                    Thread.sleep(PRODUCER_DELAY_MS);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        new Thread(producer).start();

        // 创建一个消费者线程
        Runnable consumer = () -> {
            while (true) {
                try {
                    // 在队列为空时阻塞
                    String element = queue.take();
                    System.out.println("消费了一个元素，队列中元素个数：" + queue.size());
                    Thread.sleep(CONSUMER_DELAY_MS);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
```

```
39          };
40          new Thread(consumer).start();
41      }
42  }
```

生产者 - 消费者模式的优点

支持异步处理

场景：用户注册后，需要发注册邮件和注册短信。传统的做法有两种 1.串行的方式；2.并行方式

引入消息队列，将不是必须的业务逻辑异步处理

解耦

场景：用户下单后，订单系统需要通知库存系统扣减库存。

可以消除生产者生产与消费者消费之间速度差异

在计算机当中，创建的线程越多，CPU进行上下文切换的成本就越大，所以我们在编程的时候创建的线程并不是越多越好，而是适量即可，采用生产者和消费者模式就可以很好的支持我们使用适量的线程来完成任务。

如果在某一段业务高峰期的时间里生产者“生产”任务的速率很快，而消费者“消费”任务速率很慢，由于中间的任务队列的存在，也可以起到缓冲的作用，我们在使用MQ中间件的时候，经常说的削峰填谷也就是这个意思。

过饱问题解决方案

在实际生产项目中会有些极端的情况，导致生产者/消费者模式可能出现过饱的问题。单位时间内，生产者生产的速度大于消费者消费的速度，导致任务不断堆积到阻塞队列中，队列堆满只是时间问题。

思考：是不是只要保证消费者的消费速度一直比生产者生产速度快就可以解决过饱问题？

我们只要在业务可以容忍的最长响应时间内，把堆积的任务处理完，那就不算过饱。

> 什么是**业务容忍的最长响应时间**?
> 比如埋点数据统计前一天的数据生成报表，第二天老板要看的，你前一天的数据第二天还没处理完，那就不行，这样的系统我们就要保证，消费者在24小时内的消费能力要比生产者高才行。

场景一：消费者每天能处理的量比生产者生产的少；如生产者每天1万条，消费者每天只能消费5千条。

解决办法：消费者加机器

原因：生产者没法限流，因为要一天内处理完，只能消费者加机器

场景二：消费者每天能处理的量比生产者生产的多。系统高峰期生产者速度太快，把队列塞爆了

解决办法：适当的加大队列

原因：消费者一天的消费能力已经高于生产者，那说明一天之内肯定能处理完，保证高峰期别把队列塞满就好

场景三：消费者每天能处理的量比生产者生产的多。条件有限或其他原因，队列没法设置特别大。系统高峰期生产者速度太快，把队列塞爆了

解决办法：生产者限流

原因：消费者一天的消费能力高于生产者，说明一天内能处理完，队列又太小，那只能限流生产者，让高峰期塞队列的速度慢点