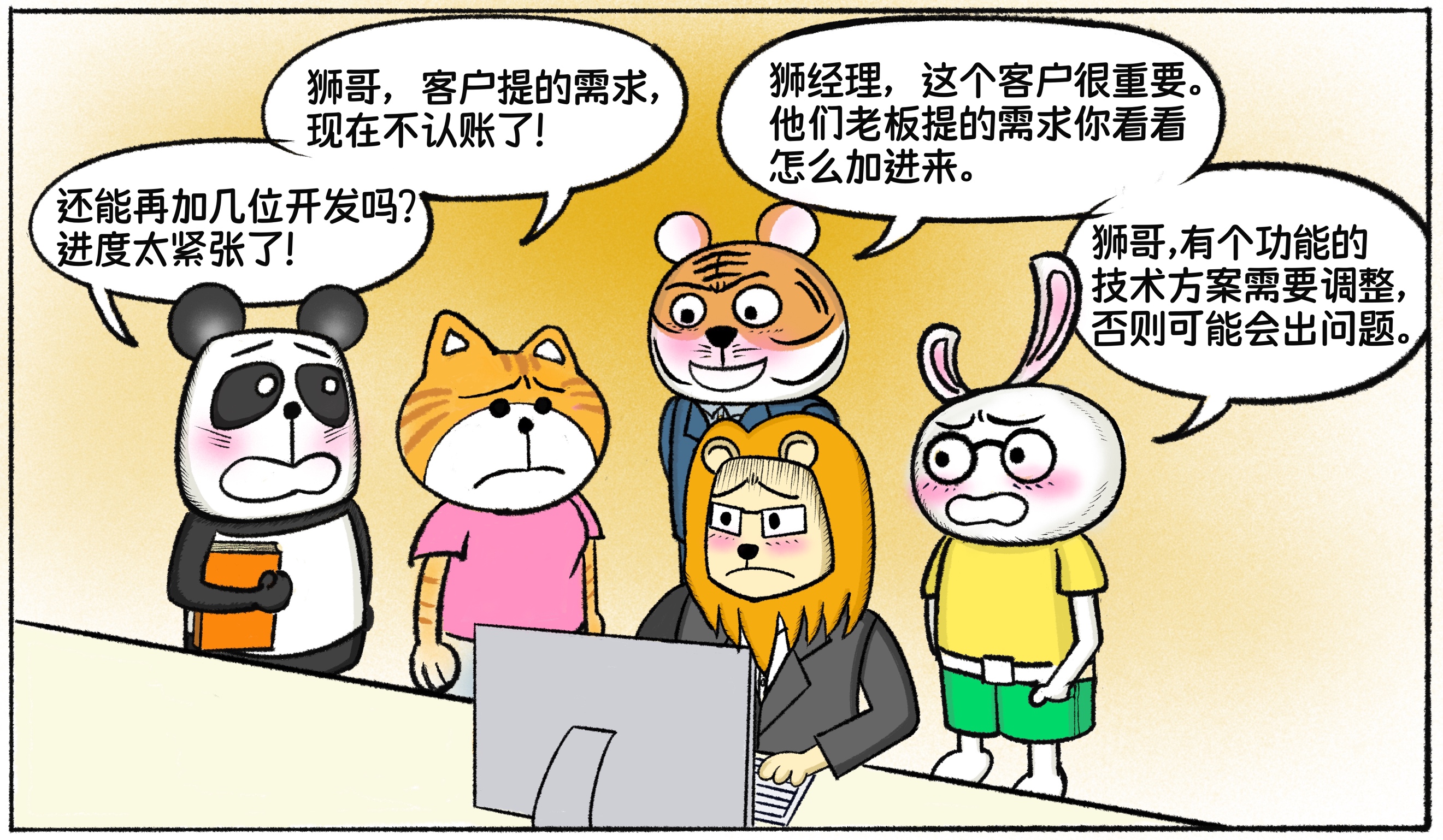
# 第8章 干活全靠我一人——单例模式

## 8.1 异常忙碌的项目经理

兔小白：我参与的第一个项目终于要上线了！最近项目经理可是忙得团团转，这两个星期一直在加班。

熊小猫：是呀，项目快要上线了，事情比较多。开发工作在收尾，客户又提了一些需求变更，经过评估也要做，进度面临很大压力。上线前还要做安全扫描，修改安全问题。UAT[[1]](#footnote-1)的问题也需要和客户沟通。最要命的是项目二期已经开始准备了。项目经理最近确实忙得不可开交。没事，你狮哥他挺得住！



兔小白：咱们公司太抠门了，就不能再安排一位项目经理吗？

熊小猫：这可不是公司抠门。一个项目一般只能有一位项目经理。项目经理需要掌握项目的所有上下文，统筹大局。咱们项目运行到这个阶段，项目上下文非常复杂，即使再派一位项目经理，也只能做一些辅助工作，作用有限，搞不好还会出乱子。项目总会有张有弛，关键时刻只能项目经理扛一下。

兔小白：还真的是这样，一个项目中，产品经理、程序员、测试人员可以有很多，但是项目经理通常只有一个。

熊小猫：程序中的对象也有同样情况，有时全局只能存在一个实例。例如程序中负责协调工作的对象，需要了解程序运行的上下文和状态，统一安排资源。如果出现两个协调者，就会产生冲突。

兔小白：如果有两个项目经理，他俩没有协调好，一个让我开发新需求，一个让我改Bug，我可就抓狂了！

熊小猫：程序里也有类似场景。例如线程池管理器要全局唯一，才能控制好线程数量。否则两个线程池管理器各管各的，线程数量就要翻倍了。

兔小白：协调工作确实只能由一个实例来承担。程序是如何做到一个类在全局只生成一个实例呢？

熊小猫：单例模式就是用来解决这个问题的。单例模式的目的很明确，确保一个类在全局只有一个实例。客户端中任何使用该类实例的地方，获取到的都是全局唯一的实例。单例模式的代码实现有一些技巧，咱们打开电脑，边写边讲。

## 8.2 懒汉式实现单例模式

熊小猫：你觉得要保证一个类只有一个实例，首先应该做什么？

兔小白：应该先把构造对象的大门要关起来，也就是不能随意通过new关键字构造对象。

熊小猫：没错，单例类要禁止使用new关键字实例化，构造方法不能被暴露。再另外提供获取单例对象的方法。

咱们看看程序如何实现。我就拿项目经理举例吧，用单例模式实现项目经理类。

public class ProjectManger {  
    private static ProjectManger instance;  
​  
    private ProjectManger() {  
   }  
​  
    public static ProjectManger getInstance(){  
        if(instance==null){  
            instance = new ProjectManger();  
       }  
​  
        return instance;  
   }  
}

熊小猫：首先使用private 修饰构造方法，对外不再提供new关键字构造ProjectManager对象的操作。getInstance方法是客户端访问ProjectManager实例的唯一方法。这是一个静态方法，可以通过类直接访问。为了确保ProjectManager实例唯一，该方法首先判断是否已经创建了单例实例。如果已经创建，则直接返回该实例，否则先创建示例再返回。

我们再来看一下客户端代码，代码中声明了两个项目经理对象，这两个对象其实指向了同一个实例。代码中对这两个对象的等值判断可以证明这一点。

ProjectManger zhangsan = ProjectManger.getInstance();  
ProjectManger lisi = ProjectManger.getInstance();  
​  
if (zhangsan == lisi) {  
   System.out.println("两位项目经理对象是同一实例");  
}

兔小白：程序中任何需要使用ProjectManager对象的地方，只能通过getInstance方法获取实例。实现 getInstance方法是确保单例的关键呀。

熊小猫：确实如此，这版代码在单线程场景下，可以确保单例。但在多线程场景下，这种实现方式存在漏洞。问题出在判断instance是否为null上。当多个线程同时执行到这一语句时，都会得到true的结果。每个线程继续向下执行，都会创建实例，从而导致创建多个实例。

兔小白：早就听说多线程编程容易出问题，原来问题出在这里！

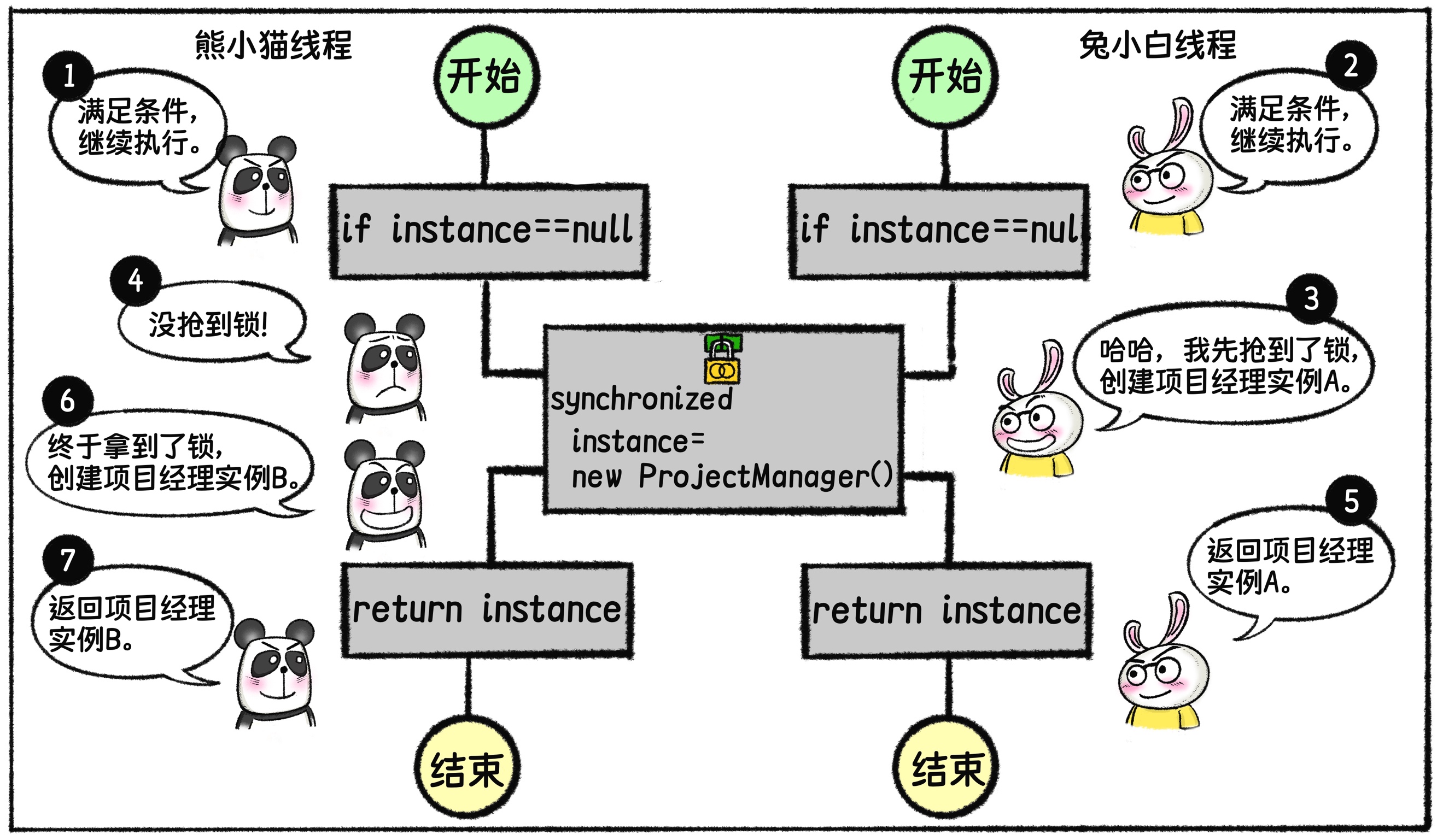
熊小猫：解决多线程并发的问题也很简单，加上synchronized代码块，就能确保同一时间只会有一个线程执行创建单例实例的代码。此外，还需要使用volatile修饰instance变量。volatile关键字用来解决多线程开发中的指令重排和变量可见性问题。这两个问题会导致getInstance方法返回其它线程还未初始化完成的instance实例。咱们看代码。

public class ProjectManger {  
    private static volatile ProjectManger instance;  
​  
    private ProjectManger() {  
   }  
​  
    public static ProjectManger getInstance() {  
        if (instance == null) {  
            synchronized (ProjectManger.class) {  
                if (instance == null) {  
                    instance = new ProjectManger();  
               }  
           }  
       }  
​  
        return instance;  
   }  
}

熊小猫：由于synchronized代码块是串行执行，存在性能问题，所以在进入synchronized代码块之前先进行一次instance是否为null的判断，减少synchronized代码块被执行的可能性。毕竟只有在第一次执行getInstance方法时，instance才可能为null。一旦instance创建完成，synchronized代码块就不会被再次执行。

兔小白：第一次判断instance是否为null，我听明白了。但是为什么在synchronized代码块中又判断了一次instance是否为null呢？

熊小猫：当两个线程并发执行时，都会通过第一次instance是否为null的判断。当执行到synchronized代码块时，其中一个线程先获取锁继续执行，创建instance实例。另一个线程锁等待锁被释放后，会获取锁继续执行synchronized代码块。如果不再做一次instance是否为null的判断，将会再次创建新的实例。下图详细展示了这个过程。



兔小白：我是真的没想到，确保只有一个实例这么难！

熊小猫：因为要考虑多线程并发的问题嘛！以后你做多线程开发的时候可一定要小心。这种单例模式的实现方式叫做懒汉式，因为程序在第一次调用getInstance方法时，才会创建单例实例。与之相对应的，还有一种叫做饿汉式的单例模式实现方式。

## 8.3 饿汉式实现单例模式

熊小猫：所谓的饿汉式，就是不管单例对象是否被用到，程序都会提前将单例实例创建好。如果不担心内存占用，可以考虑采用饿汉式实现单例模式，代码更为简洁，当然也是线程安全的。我们来看看代码实现。

public class ProjectManger {  
    private static ProjectManger instance = new ProjectManger();  
​  
    private ProjectManger() {  
   }  
​  
    public static ProjectManger getInstance(){  
        return instance;  
   }  
}

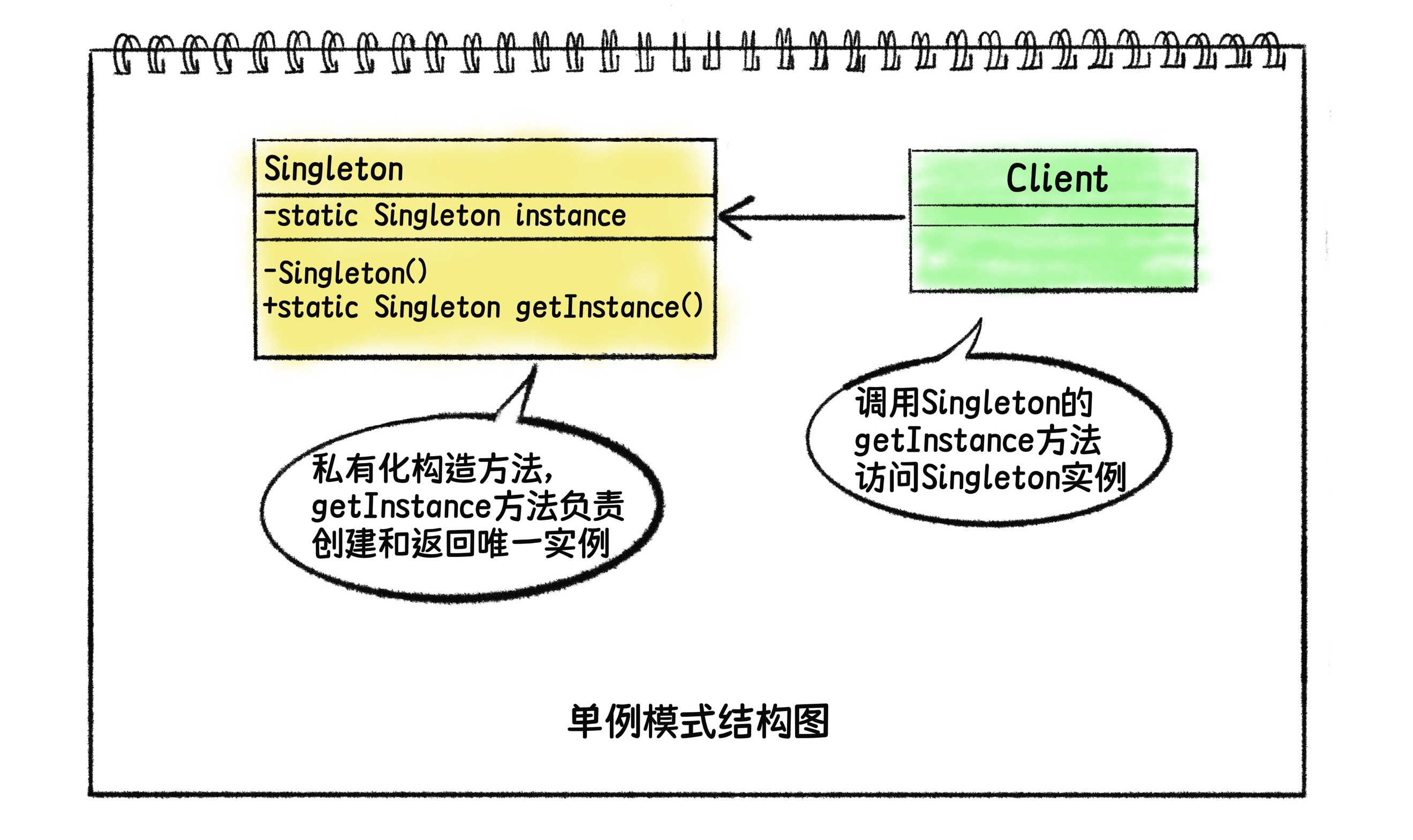
熊小猫：类加载时会完成静态成员变量的初始化，因此只要类加载完，就会完成instance变量的初始化。创建单例实例发生在类加载的过程中，因此不存在线程安全问题。当客户端调用getInstance方法时，可以直接获取到已经创建好的实例。

兔小白：这种实现方式简洁多了！咱们已经讲了三种单例模式的实现方式，你推荐采用哪一种呢？

熊小猫：咱们讲的这三种实现方式都很常用，但各自有其适合的场景。懒汉式实现方式，在客户端第一次调用getInstance方法时创建实例，避免了内存的浪费。选择哪一种懒汉式实现方式，取决于是否存在多线程场景。饿汉式实现方式的代码简洁、线程安全，但不管是否使用单例对象，程序都会提前在类加载时创建实例，占用内存。如果对内存的使用没有严格限制，推荐使用饿汉式实现方式。

## 8.4 单例模式适用场景

熊小猫：单例模式非常简单，我们看下面的单例模式结构图。



熊小猫：单例模式虽然简单，但有着独特的适用场景，如果使用不当会引起线程安全问题。我们先来看看单例模式的优点。

1. 单例类在内存中只存在一个实例，减少了内存的使用。
2. 单例类的实例化严格受控。客户端只能通过单例类提供的方法访问单例实例。统一出口，便于管理。
3. 线程间共享单例实例的成员变量，适合中心化工作。

兔小白：最后一个优点，如果使用不当会出现线程安全的问题啊！

熊小猫：没错，多线程的场景下，使用单例模式一定要小心。我们再来看看单例模式的缺点。

1. 构造单例实例的逻辑在单例类内部实现，和单一职责原则存在冲突。
2. 单例实例的成员变量被线程共享。处理不当会引发线程安全问题。例如游戏人物的血量属性。程序必须保证对血量的处理是线程安全的，否则当同时受到不同来源的攻击时，血量计算会出错。

可以看到单例模式的优、缺点都很鲜明。单例模式在与之匹配的场景下才能发挥它的优势。如使用不当，反而会出现线程安全问题。下面是适合使用单例模式的场景。

1. 中心化工作场景。例如负责全局协调、对象管理、资源调度的类，需要保证它的实例全局唯一，否则对资源的管理和使用无法做到同步。
2. 没有成员变量的类适合使用单例模式。如果类没有任何成员变量，一是创建多个实例没有意义，二是单例实例也不存在线程安全问题。这个场景比较常见的是工具类，使用静态类来实现也能达到同样的效果。

使用单例模式时，特别需要注意一点，虽然单例模式能够节省内存开销，但这只是它的附加价值，不应该为了节省内存而去刻意使用单例模式，否则会带来更为麻烦的线程安全问题。使用单例模式时，一定要注意多线程对单例对象成员变量的访问，做好线程同步。

兔小白：我真的没想到单例模式这么复杂，里面居然有这么多学问。

熊小猫：由于牵扯到多线程，问题的复杂度会升高一个等级。多线程编程里面的学问可不少呢！等学习完设计模式，咱们再深入研究多线程编程！

1. UAT全称是User Acceptance Test，即用户验收测试。 [↑](#footnote-ref-1)