

Professor Hans Noodles

41 уровень

24.10.2019 35570 24

Паттерны проектирования: Singleton

Статья из группы **Java Developer**
43656 участников

Вы в группе

Привет! Сегодня будем подробно разбираться в разных паттернах проектирования, и начнем с шаблона Singleton, который еще называют “одиночка”.



Давай вспомним: что мы знаем о шаблонах проектирования в целом?

Шаблоны проектирования — это лучшие практики, следуя которым можно решить ряд известных проблем.

Шаблоны проектирования как правило не привязаны к какому-либо языку программирования. Воспринимай их как свод рекомендаций, следуя которым можно избежать ошибок и не изобретать свой велосипед.

Что такое синглтон?

Синглтон — это один из самых простых шаблонов (паттернов) проектирования, который применяется к классу. Иногда говорят: “этот класс — синглтон”, подразумевая, что этот класс реализует паттерн проектирования синглтон.

Иногда необходимо написать класс, у которого можно будет создать только один объект. Например, класс, отвечающий за логирование или подключение к базе данных.

Шаблон проектирования синглтон описывает, как мы можем выполнить такую задачу.

Синглтон — это шаблон (паттерн) проектирования, который делает две вещи:

НАЧАТЬ ОБУЧЕНИЕ

2. Предоставляет глобальную точку доступа к экземпляру данного класса.

Отсюда — две особенности, характерные для практически каждой реализации паттерна синглтон:

1. Приватный конструктор. Ограничивает возможность создания объектов класса за пределами самого класса.
2. Публичный статический метод, который возвращает экземпляр класса. Данный метод называют `getInstance`. Это глобальная точка доступа к экземпляру класса.

Варианты реализации

Шаблон проектирования синглтон применяют по-разному. Каждый вариант по-своему хорош и плох. Тут как всегда: идеала нет, но нужно к нему стремиться.

Но прежде всего давай определимся, что такое хорошо и что такое плохо, и какие метрики влияют на оценку реализации шаблона проектирования.

Начнем с положительного. Вот критерии, которые придают реализации сочности и привлекательности:

- Ленивая инициализация: когда класс загружается во время работы приложения именно тогда, когда он нужен.
- Простота и прозрачность кода: метрика, конечно, субъективная, но важная.
- Потокобезопасность: корректная работа в многопоточной среде.
- Высокая производительность в многопоточной среде: потоки блокируют друг друга минимально, либо вообще не блокируют при совместном доступе к ресурсу.

Теперь минусы. Перечислим критерии, которые выставляют реализацию в нелучшем свете:

- Не ленивая инициализация: когда класс загружается при старте приложения, независимо от того, нужен он или нет (парадокс, в мире IT лучше быть лентяем)
- Сложность и плохая читаемость кода. Метрика также субъективная. Будем считать, что если кровь пошла из глаз, реализация так себе.
- Отсутствие потокобезопасности. Иными словами, “потокопасность”. Некорректная работа в многопоточной среде.
- Низкая производительность в многопоточной среде: потоки блокируют друг друга все время либо часто, при совместном доступе к ресурсу.

Научитесь программировать с нуля с JavaRush:
1200 задач, автопроверка решения и стиля кода

НАЧАТЬ ОБУЧЕНИЕ

Код

Теперь мы готовы рассмотреть различные варианты реализации с перечислением плюсов и минусов:

Simple Solution

```
1 public class Singleton {
2     private static final Singleton INSTANCE = new Singleton();
3
4     private Singleton() {
5     }
6
7     public static Singleton getInstance() {
```

НАЧАТЬ ОБУЧЕНИЕ

```
10         }
11     }
12 }
```

Самая простая реализация.

Плюсы:

- Простота и прозрачность кода
- Потокобезопасность
- Высокая производительность в многопоточной среде

Минусы:

- Не ленивая инициализация.

В попытке исправить последний недостаток, мы получаем реализацию номер два:

Lazy Initialization

```
1  public class Singleton {
2      private static Singleton INSTANCE;
3
4      private Singleton() {}
5
6      public static Singleton getInstance() {
7          if (INSTANCE == null) {
8              INSTANCE = new Singleton();
9          }
10         return INSTANCE;
11     }
12 }
```

Плюсы:

- Ленивая инициализация.

Минусы:

- Не потокобезопасно

Реализация интересна. Мы можем инициализироваться лениво, но утратили потокобезопасность. Не беда: в реализации номер три мы все синхронизируем.

Synchronized Accessor

```
1  public class Singleton {
2      private static Singleton INSTANCE;
3
4      private Singleton() {
5      }
6
7      public static synchronized Singleton getInstance() {
8          if (INSTANCE == null) {
```

```
11         return INSTANCE;
12     }
13 }
```

Плюсы:

- Ленивая инициализация.
- Потокобезопасность

Минусы:

- Низкая производительность в многопоточной среде

Отлично! В реализации номер три мы вернули потокобезопасность! Правда, медленную... Теперь метод `getInstance` синхронизирован, и входить в него можно только по одному.

На самом деле нам нужно синхронизировать не весь метод, а лишь ту его часть, в которой мы инициализируем новый объект класса. Но мы не можем просто обернуть в `synchronized` блок часть, отвечающую за создание нового объекта: это не обеспечит потокобезопасность. Все немного сложнее.

Правильный способ синхронизации представлен ниже:

Double Checked Locking

```
1 public class Singleton {
2     private static Singleton INSTANCE;
3
4     private Singleton() {
5     }
6
7     public static Singleton getInstance() {
8         if (INSTANCE == null) {
9             synchronized (Singleton.class) {
10                 if (INSTANCE == null) {
11                     INSTANCE = new Singleton();
12                 }
13             }
14         }
15         return INSTANCE;
16     }
17 }
```

Плюсы:

- Ленивая инициализация.
- Потокобезопасность
- Высокая производительность в многопоточной среде

Минусы:

- Не поддерживается на версиях Java ниже 1.5 (в версии 1.5 исправили работу ключевого слова `volatile`)

Отмечу, что для корректной работы данного варианта реализации обязательно одно из двух условий. Переменная `INSTANCE` должна быть либо `final`, либо `volatile`.

Последняя реализация, которую мы сегодня обсудим, — `Class Holder Singleton`.

Class Holder Singleton

```
1 public class Singleton {
2
3     private Singleton() {
4     }
5
6     private static class SingletonHolder {
7         public static final Singleton HOLDER_INSTANCE = new Singleton();
8     }
9
10    public static Singleton getInstance() {
11        return SingletonHolder.HOLDER_INSTANCE;
12    }
13 }
```

Плюсы:

- Ленивая инициализация.
- Потокобезопасность.
- Высокая производительность в многопоточной среде.

Минусы:

- Для корректной работы необходима гарантия, что объект класса `Singleton` инициализируется без ошибок. Иначе первый вызов метода `getInstance` закончится ошибкой `ExceptionInInitializerError`, а все последующие `NoClassDefFoundError`.

Реализация практически идеальная. И ленивая, и потокобезопасная, и быстрая. Но есть нюанс, описанный в минусе.

Сравнительная таблица различных реализаций паттерна Singleton:

Реализация	Ленивая инициализация	Потокобезопасность	Скорость работы при многопоточности	Когда использовать?
Simple Solution	-	+	Быстро	Никогда. Либо когда не важна ленивая инициализация. Но лучше никогда.
Lazy Initialization	+	-	Неприменимо	Всегда, когда не нужна многопоточность
Synchronized Accessor	+	+	Медленно	Никогда. Либо когда скорость работы при многопоточности не имеет значения. Но лучше никогда
Double Checked Locking	+	+	Быстро	В редких случаях, когда нужно обрабатывать исключения при создании синглтона. (когда неприменим Class Holder Singleton)

Реализация	Ленивая инициализация	Потокобезопасность	Скорость работы при многопоточности	Когда использовать?
Class Holder Singleton	+	+	Быстро	Всегда, когда нужна многопоточность и есть гарантия, что объект синглтон класса будет создан без проблем.

Плюсы и минусы паттерна Singleton

В целом синглтон делает именно то, что от него ждут:

1. Дает гарантию, что у класса будет всего один экземпляр класса.
2. Предоставляет глобальную точку доступа к экземпляру данного класса.

Однако у этого шаблона есть недостатки:

1. Синглтон нарушает SRP (Single Responsibility Principle) — класс синглтона, помимо непосредственных обязанностей, занимается еще и контролированием количества своих экземпляров.
2. Зависимость обычного класса или метода от синглтона не видна в публичном контракте класса.
3. Глобальные переменные это плохо. Синглтон превращается в итоге в одну здоровенную глобальную переменную.
4. Наличие синглтона снижает тестируемость приложения в целом и классов, которые используют синглтон, в частности.

Ну вот и все. Мы рассмотрели с тобой паттерн проектирования синглтон. Теперь в разговоре за жизнь с друзьями программистами ты сможешь сказать не только чем он хорош, но и пару слов о том, чем он плох.

Удачи в освоении новых знаний.

Дополнительное чтение:

- [Использование паттерна синглтон](#)

−

+113

+

Комментарии (24)

популярные

новые

старые

JavaCoder

Введите текст комментария

Rostik Уровень 36, Москва, Russian Federation

20 апреля, 16:43

⋮

Касаемо раздела статьи "Double Checked Locking". Автор забыл обязательное в данном случае ключевое слово volatile. [Смотрим](#), почему. Да и в принципе, guys, советую целиком чекнуть данное видео. Довольно чётко & ясно про азы multithreading.

Ответить

−

+4

+

CyberBoar Уровень 39, Russian Federation

12 апреля, 09:18

⋮

И как в последнем случае нам гарантировать, что объект инициализируется без ошибок?

Ответить

−

+1

+

Denis Уровень 32, Краснодар, Russian Federation

22 декабря 2021, 07:47

⋮

НАЧАТЬ ОБУЧЕНИЕ

Так же я помню из других источников, что инициализация статических полей и статических блоков класса выполняется ОДИН РАЗ ПРИ ПЕРВОМ ОБРАЩЕНИИ К КЛАССУ.
Получается, что если разработчик в своём коде не обращается к объекту класса Singleton, то и статические поля и статические блоки выполнятся не будут, а значит и не будет создан экземпляр класса. Тогда единственный минус первого примера как бы отпадает - верно?
А значит первый вариант - самый простой и правильный вариант паттерна Singleton.
Кто может объяснить мне, почему в статье в самом первом примере указан минус паттерна "Не ленивая инициализация"?

Ответить

+2

fedyaka Уровень 36, Кострома, Россия

12 января, 19:26

На сколько я понимаю, здесь скорей Simple Solution непотокобезопасное, из за того что к классу может обратиться несколько потоков одновременно и начать создавать новые объекты, и записывать их в переменную(ошибка, ведь это переменная final, перезаписать нельзя). Либо пока первый поток будет создавать объект, другой увидя что объект есть(но ещё не доделанный), попыбует его достать и получит непонятную консистенцию. Ведь ничего не мешает им это сделать, синхронизации никакой нет.

Но это лишь мои догадки, утверждать не могу.

Ответить

0

Игорь Full Stack Developer в IgorApplications

9 августа 2021, 15:35

Никого не смущает последний - Class Holder Singleton, якобы лучший пример? Зачем понадобился вложенный класс, если получается тоже самое, что и в самом первом примере.

Ответить

0

Игорь Уровень 33, Москва , Россия

13 октября 2021, 08:53

Я так понимаю у первого варианты минус отсутсвие ленивой инициализации(своими словами, в первом случае объект инициализируется в момент компиляции). В последнем варианте, инициализация происходит в момент вызова метода getInstance, статический объект класса в данном случае создаётся в момент вызова из публичного.
Также почитав об этом паттерне... По сути сами разработчики Java рекомендуют создавать этот паттерн через enum
public enum Singleton{
 INSTANCE
}

Ответить

0

Владимир Лукашов Superman

19 июля 2021, 09:25

```
1  if (INSTANCE == null) INSTANCE = new SingletonExample1();
2
3      return INSTANCE;
```

Так Singleton работает правильно
а если написать вот так

```
1  return (INSTANCE == null) ? new SingletonExample1() : INSTANCE;
```

то создается новый объект каждый раз
SingletonDemo{
example1=Singleton.SingletonExample1@67b64c45
}
SingletonDemo{
example1=Singleton.SingletonExample1@4411d970
}
SingletonDemo{
example1=Singleton.SingletonExample1@6442b0a6
}
SingletonDemo{
example1=Singleton.SingletonExample1@60f82f98
}
По факту суть кода одна но результат разный.
Есть люди кто разобрался в этом? Я че то догнать не могу

Ответить

0

Диана Чиганцева Уровень 39, Краснодар, Россия

26 августа 2021, 15:58

Где во втором случае тот момент когда, если INSTANCE == null, то мы инициализируем ее объектом SingletonExample1?

Ответить

+1

Владимир Лукашов Superman

26 августа 2021, 17:13

Догнал)
Во втором случае мы возвращаем новый объект, но не сохраняем ссылку на него в INSTANCE.
Поэтому INSTANCE каждый раз будет null и всегда будет создаваться новый объект

Ответить

0

НАЧАТЬ ОБУЧЕНИЕ

синглтона занимает какое-то время, то остальным потокам придется подождать, но точно также им нужно будет подождать и в "улучшенной" версии перед входом в synchronized блок. При последующих обращениях метод будет захватывать монитор, проверять что ссылка не null и возвращать объект. Но ведь это всего лишь пара тактов процессора, что мы потеряем, 5 наносекунд? По сути к каким-то заметным проблемам это может привести только если разным потокам десятки и сотни тысяч раз пришло в голову вызывать метод getInstance(). Или я не прав?

Ответить

−

+2

+

LeonidJava Developer в ProgForceEXPERT13 июня 2020, 15:01

...

Почему в варианте Class Holder Singleton внутренний сласс public а не private?

```
1 public static class SingletonHolder {
2     public static final Singleton HOLDER_INSTANCE = new Singleton();
3 }
```

Ответить

−

+1

+

AlukardVampire hunter в The HellsingEXPERT31 мая 2020, 18:26

...

Что я тут забыл на 14м уровне?)

Ответить

−

0

+

Marianna PatrusovaУровень 0, Минск, Беларусь30 мая 2020, 14:26

...

Когда проходили пул соединений, препод советовал делать синглтон через enum или использовать вместо synchronized ReentrantLock'и. Не знаю, правда, насколько это производительно. Кто-нибудь подскажет?

кусочек класса:

```
public class ConnectionPool {

    private static Lock lock = new ReentrantLock();
    private static AtomicBoolean flag = new AtomicBoolean(false);
    private static ConnectionPool instance;
    //ещё переменные

    private ConnectionPool() {
        if (instance != null) {
            //защита от рефлексии
            throw new RuntimeException("Attempt to create second pool's instance.");
        }
    }

    public static ConnectionPool getInstance() {
        if (!flag.get()) {
            try {
                lock.lock();
                if (instance == null) {
                    instance = new ConnectionPool();
                    flag.set(true);
                }
            } finally {
                lock.unlock();
            }
        }
        return instance;
    }

    //защита от клонирования
    public final Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }

    // ещё методы
}
```

Ответить

−

0

+

SorosУровень 39, Харьков, Украина25 апреля 2020, 14:54

...

```
1 public class Singleton {
2     private static final Singleton INSTANCE;
3
4     private Singleton() {
5     }
6
7     public static Singleton getInstance() {
8         if (INSTANCE == null) {
9             synchronized (Singleton.class) {
10                 if (INSTANCE == null) {
11                     INSTANCE = new Singleton();
12                 }
13             }
14         }
15     }
16 }
```

НАЧАТЬ ОБУЧЕНИЕ


```
15         return INSTANCE;  
16     }  
17 }
```

То, что модификатор `final` здесь не уместен, и уже много раз на это указывали, но до сих пор не исправлено - не очень хорошо. Зачем в материале лекции допускается такая ошибка?

Минусом такой реализации указано "Не поддерживается на ранних версиях Java. Ниже 1.5"
А почему? Что этому коду не даёт возможность быть использованным в версиях до 1.5?

Ответить

−

+2

+

Leonid Java Developer в **ProgForce** EXPERT13 июня 2020, 15:20

⋮

До java 5 не было слова `volatile` насколько я помню. Но в статье причину, разумеется, нужно было написать.

Ответить

−

+2

+

↺ Показать еще комментарии

ОБУЧЕНИЕ

- Курсы программирования
- Курс Java
- Помощь по задачам
- Подписки
- Задачи-игры

СООБЩЕСТВО

- Пользователи
- Статьи
- Форум
- Чат
- Истории успеха
- Активности

КОМПАНИЯ

- О нас
- Контакты
- Отзывы
- FAQ
- Поддержка



JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

 Русский

▼

