ZLToolKit笔记9：设计模式

2024年12月28日　15:45

有关设计模式的内容参见：C++设计模式

一 单例模式

在EventPoller.cpp中有这句代码：

```
INSTANCE_IMP(EventPollerPool)
```

这其实是个宏定义，宏定义和宏展开如下，也就是说**上面个这句代码会在预处理的时候被下面黑色的代码所替换。**

```
#define INSTANCE_IMP(class_name, ...) \
class_name &class_name::Instance() { \
    static std::shared_ptr<class_name> s_instance(new class_name(__VA_ARGS__)); \
    static class_name &s_instance_ref = *s_instance; \
    return s_instance_ref; \
}
```

扩展到：

```
EventPollerPool &EventPollerPool::Instance() {
    static std::shared_ptr<EventPollerPool> s_instance(new EventPollerPool());
    static EventPollerPool &s_instance_ref = *s_instance;
    return s_instance_ref;
}
```

这段代码实现了一个线程安全的**单例模式**，返回一个 EventPollerPool 的唯一实例。具体功能如下：

1. **单例实现**：通过 static 变量，确保 EventPollerPool 的实例在程序生命周期内只会被创建一次。
2. **共享所有权**：使用 std::shared_ptr 管理单例对象，确保生命周期由引用计数自动管理。
3. **全局访问**：Instance() 函数提供了一个全局访问点，用于获取 EventPollerPool 的实例。

函数内 static 的作用

1. static 变量的特点

- **局部静态变量**：在函数范围内声明，但具有静态存储期。
- **初始化一次**：在第一次调用 Instance() 函数时，s_instance 和 s_instance_ref 会被初始化。之后，所有对 Instance() 的调用都会返回同一个实例。
- **线程安全**：C++11 标准保证了局部静态变量的初始化是线程安全的。

2. 在代码中的具体作用

- s_instance：
  - 是一个 std::shared_ptr，持有 EventPollerPool 实例的所有权。
  - 确保实例在不再使用时可以被释放（虽然单例通常不会释放）。
- s_instance_ref：

- 是 EventPollerPool 的引用，简化了访问逻辑。
- 通过引用避免了每次调用时需要解引用 std::shared_ptr，提供更直接的访问方式。

这样设计的好处

1. **单例实现**:
   - 确保系统中只有一个 EventPollerPool 实例。
   - 适用于需要全局管理的资源，如线程池、事件循环等。
2. **线程安全性**:
   - 使用 static 变量和 C++11 标准的线程安全特性，避免了多线程环境下可能的竞争条件。
3. **内存管理**:
   - 使用 std::shared_ptr 提供自动内存管理。
   - 如果程序设计允许释放单例对象，可以通过 std::shared_ptr 的引用计数实现（尽管单例通常不需要释放）。
4. **高效访问**:
   - s_instance_ref 提供了直接访问实例的方式，避免频繁解引用 std::shared_ptr，提高了运行效率。
5. **懒加载**:
   - 单例对象的实例化延迟到第一次调用 Instance() 函数时，而不是程序启动时。这节省了资源，尤其是在对象初始化代价较大且可能不被使用的情况下。

函数内 static 修饰的变量

在 C++ 中，static 修饰的局部变量具有特殊的存储和生命周期特性。以下是其详细特点和作用:

1. 特点

1. **存储期：静态存储期**
   - 函数内 static 变量在程序运行期间**仅被初始化一次**，并且**一直存在于内存中**直到程序结束。
   - 即使函数调用结束，static 变量的值也不会销毁或重置。
2. **作用域：局部作用域**
   - 变量只能在函数内部访问，外部无法直接使用，符合局部变量的作用域规则。
3. **初始值:**
   - 默认会被初始化为其类型的默认值（例如 int 默认初始化为 0，指针默认初始化为 nullptr）。
   - 如果用户显式指定初值，则在第一次函数调用时进行初始化。
4. **线程安全（C++11 及更高版本）**
   - 在多线程环境中，C++11 保证了局部 static 变量的初始化是线程安全的，多个线程不会导致竞争条件。

2. 使用场景

1. **函数内的持久性数据存储**:
   - 用于在函数中保存需要跨调用的状态信息。
   - 例如计数器、缓存、配置数据等。

2. **实现单例模式的局部单例**
    - 局部 static 变量可以在函数内部创建一个全局唯一对象，常用于实现单例模式。
3. **延迟初始化**:
    - 静态局部变量只有在函数第一次调用时才会被初始化，适用于某些初始化成本较高的对象或在运行时才能确定的值。

3. 示例分析

示例 1：计数器

cpp
复制代码
```
int counter() {
    static int count = 0; // 静态局部变量，只初始化一次
    return ++count;       // 每次调用，值会递增
}
int main() {
    std::cout << counter() << std::endl; // 输出: 1
    std::cout << counter() << std::endl; // 输出: 2
    std::cout << counter() << std::endl; // 输出: 3
}
```

- static int count 在第一次调用 counter() 时被初始化为 0。
- 在后续的函数调用中，count 保持其值，不会被重新初始化。

示例 2：单例实现

cpp
复制代码
```
class Singleton {
public:
    static Singleton& getInstance() {
        static Singleton instance; // 静态局部变量，线程安全
        return instance;
    }
private:
    Singleton() {} // 私有构造函数，防止外部实例化
};
```

- 通过 static 确保 Singleton 的实例在全局范围内唯一。
- 延迟初始化：实例只有在第一次调用 getInstance 时才会被创建。

4. 静态局部变量与全局变量的对比

特性	静态局部变量	全局变量
作用域	函数内（局部作用域）	全局
存储期	程序运行期间（静态存储期）	程序运行期间（静态存储期）
初始化时机	第一次调用所在函数时	程序启动时

访问限制 只能在声明的函数内访问 任何地方都可访问

封装性 高（局部变量的特性） 低（容易被全局修改）

5. 优缺点

优点：

1. **高效：**
   - 初始化只执行一次，减少了不必要的重复工作。
2. **作用域限定：**
   - 与全局变量相比，作用域受到限制，提高了封装性和代码可维护性。
3. **线程安全：**
   - 从 C++11 开始，局部静态变量的初始化是线程安全的。

缺点：

1. **难以测试：**
   - static 变量隐藏在函数内，可能导致测试覆盖不足。
2. **资源占用：**
   - 无法主动释放，可能导致长期运行程序中占用不必要的资源。

6. 总结

函数内 static 修饰的变量，是一种非常实用的设计模式。它可以在局部范围内提供全局性持久化存储，同时限制访问范围，提高封装性。常用于计数器、缓存、延迟初始化和单例模式的实现。

将静态变量 s_instance 定义在函数 Instance 内部，而不是作为 EventPollerPool 类的私有成员变量，有以下好处和意义：

1. 延迟初始化

- **函数内部 static：**
  - 静态局部变量只有在 Instance 函数第一次被调用时才会被初始化。这种**"懒加载"**的方式避免了在程序启动时就初始化，而是将初始化延迟到真正需要使用时，节省了资源。
  - 适用于单例模式中实例的延迟创建，尤其当实例化成本较高或者实例可能并不一定总会被使用时。
- **作为类成员：**
  - 如果是类的私有静态成员，初始化时机通常在程序启动时完成（如果静态成员定义为直接初始化），即使该对象可能永远不会被使用，仍然会占用资源。

2. 提高封装性

- **函数内部 static：**
  - s_instance 被限制在 Instance 函数内部，外界完全无法直接访问它，保证了更高的封装性。只有通过 Instance 函数，才能访问或创建该单例对象。
  - 避免了静态成员暴露在类中时的潜在滥用风险。
- **作为类成员：**
  - 即使是私有静态成员变量，仍然是类的一部分，可能会被其他类的成员函数通过友元或反射等机制访

问到，封装性稍弱。

3. 更简洁的代码

- **函数内部 static：**
  - 静态局部变量的定义和初始化合并在 Instance 函数中，代码更加紧凑且直观。
  - 不需要在类外部为静态成员变量显式定义和初始化（C++17 以前，静态成员变量需要类外定义）。
- **作为类成员：**
  - 静态成员变量需要在类外部提供定义（非内联变量），代码结构分散，显得冗余。

4. 线程安全性

- **函数内部 static：**
  - 从 C++11 开始，函数内的静态局部变量初始化是 **线程安全** 的。这意味着即使多个线程同时调用 Instance，编译器也会确保 s_instance 的初始化只发生一次。
- **作为类成员：**
  - 静态成员变量的线程安全初始化需要开发者手动实现（如使用 std::call_once 等机制），增加了代码复杂性和错误可能性。

5. 防止非预期的静态初始化顺序问题

- **函数内部 static：**
  - 局部静态变量的初始化受限于函数调用，不会受到其他静态对象初始化顺序的影响。
  - 解决了静态变量初始化可能依赖其他静态变量的问题（静态初始化顺序问题）。
- **作为类成员：**
  - 静态成员变量初始化可能在程序启动时就进行，若它依赖于其他静态对象，可能会导致未定义行为。

6. 提供更灵活的使用方式

- **函数内部 static：**
  - 这种设计使得单例的生命周期完全由 Instance 函数的调用管理，符合单例模式的设计原则。
- **作为类成员：**
  - 静态成员变量的生命周期绑定到类，难以实现延迟销毁或自定义生命周期控制。

7. 示例对比

函数内局部静态变量实现

cpp
复制代码
```
class EventPollerPool {
public:
    static EventPollerPool& Instance() {
        static EventPollerPool s_instance; // 延迟初始化，线程安全
        return s_instance;
    }
private:
    EventPollerPool() = default;
```

```
};
```

类内静态成员变量实现

```
cpp
复制代码
class EventPollerPool {
public:
    static EventPollerPool& Instance() {
        return s_instance; // 返回已有的静态成员
    }
private:
    static EventPollerPool s_instance; // 必须类外定义
    EventPollerPool() = default;
};
// 类外定义
EventPollerPool EventPollerPool::s_instance; // 初始化
```

总结

将 s_instance 定义为 Instance 函数中的静态局部变量，而不是类的静态成员变量，主要有以下优势：

1. 延迟初始化，节省资源。
2. 提高封装性，减少外界干扰。
3. 避免静态初始化顺序问题。
4. 利用 C++11 提供的线程安全特性，代码简洁可靠。
5. 减少静态成员的复杂性，代码更清晰。

这种设计符合现代 C++ 的单例模式实现风格，是一种更优雅和高效的方式。

一 工厂模式

1. 简单工厂模式
2. 工厂方法模式
3. 抽象工厂模式

工厂模式简介

工厂模式是一种创建对象的设计模式，通过定义一个接口或抽象类，让子类或具体的实现类来决定对象实例化的方式。工厂模式主要分为以下几种类型：

1. **简单工厂模式**：一个工厂类根据传入的参数决定创建哪种具体的对象。
2. **工厂方法模式**：将对象的创建推迟到子类中，使用子类实现来创建具体的产品对象。
3. **抽象工厂模式**：提供一个接口，允许创建一系列相关或依赖的对象，而无需指定它们的具体类。

工厂模式的目的

- **解耦对象创建和使用**：将实例化逻辑与具体使用逻辑分离，提高代码可维护性。
- **便于扩展**：新增对象类型时，只需修改工厂逻辑，而不影响其他代码。
- **简化对象管理**：可以对创建的对象进行统一管理，如复用、缓存等。

在 ZLToolKit 项目中的体现

BufferRaw 类及其 create 函数就是工厂模式的一种实现，以下结合代码具体说明其使用了哪些工厂模式思想：

代码分析

1. **创建接口统一化**
   - 静态方法 BufferRaw::create() 提供了创建 BufferRaw 对象的接口，屏蔽了具体实现细节。
   - 调用者通过 create() 获取实例，无需关心 BufferRaw 的构造方式或内存分配细节。
2. **分离创建和使用**
   - BufferRaw 提供了 assign、setCapacity 等功能接口，用于操作数据，但创建过程通过 create 统一管理。
   - 调用者只需依赖 BufferRaw::Ptr，不需要直接管理 BufferRaw 的生命周期。
3. **支持对象池优化**
   - create 函数的 #if 0 部分展示了一种基于对象池的实现：
     - 使用 ResourcePool<BufferRaw> 提供了对 BufferRaw 对象的复用。
     - 对象池 packet_pool 提前分配了一批 BufferRaw 对象，减少了频繁创建销毁对象的开销，提高了性能。
   - 对象池的使用将资源管理与对象创建逻辑集中到 create 内部，调用者完全透明。
4. **增强扩展性**
   - 如果需要切换到更复杂的 BufferRaw 实现（如引入缓存策略或引用计数），只需修改 create 方法的实现。
   - 例如，可以通过配置开关（如 #if 0）动态调整对象创建逻辑，而不需要更改使用 BufferRaw 的代码。

工厂模式的好处

通过这种设计：

1. **统一创建接口**：BufferRaw::create 提供了一致的对象创建方式，隐藏了具体的内存管理和初始化逻辑。
2. **提升性能**：结合对象池机制，避免频繁分配和销毁内存。
3. **增强灵活性**：可以轻松替换创建逻辑，如从对象池切换到直接分配或其他复杂初始化方案。
4. **解耦依赖**：调用者只需依赖 BufferRaw::Ptr，无需关注底层实现。

对比传统设计的改进

传统设计方式

在没有工厂模式的情况下，创建 BufferRaw 通常直接调用构造函数，例如：

cpp
复制代码
```
BufferRaw *buf = new BufferRaw();
```
缺点：

1. **代码冗余**：每次都需要手动调用构造函数。

2. **生命周期管理复杂**：需要手动调用 delete，容易导致内存泄漏。
3. **扩展性差**：若创建逻辑发生变化（如引入对象池），需要修改所有调用点。

工厂模式的改进

通过 create：

cpp
复制代码
```
auto buf = BufferRaw::create();
```

优点：

1. **简洁**：只需调用工厂方法，无需关心构造细节。
2. **自动管理内存**：结合智能指针，自动释放内存，避免泄漏。
3. **易扩展**：通过修改 create 方法，可以轻松支持新特性（如对象池）。

除了以上的BufferRaw的设计结合了工厂模式的设计思想，还有下面BufferList的设计也是，其定义在ZLToolKit笔记8：Buffer设计中

从代码设计来看，BufferList::create 函数体现了**工厂方法模式（Factory Method Pattern）**，具体特点如下：

1. **抽象基类与派生类分离**：
   - BufferList 是抽象基类，定义了接口 empty、count、send，以及静态工厂方法 create。
   - 实现具体功能的逻辑由派生类 BufferSendTo、BufferSendMsg、BufferSendMMsg 提供。
2. **动态选择具体派生类**：
   - 根据系统条件（如操作系统类型）和 is_udp 参数，动态决定创建哪种派生类的实例。
3. **隐藏派生类细节**：
   - 调用者只依赖 BufferList 接口，不需要直接操作派生类。
   - 派生类的具体实现对外透明。

工厂方法模式的好处

1. **解耦创建与使用**
   - 调用者无需知道派生类 BufferSendTo、BufferSendMsg 等的存在，只需调用 BufferList::create 创建对象。
   - 对象的使用通过 BufferList 提供的统一接口完成。
2. **便于扩展**
   - 如果需要新增一种发送策略（如支持新的系统或优化 UDP 方案），只需增加一个新的派生类并修改 create 方法即可，无需改变调用逻辑。
3. **针对不同平台和条件优化**
   - create 方法根据编译条件动态选择派生类：
     - Windows 使用 sendto 或 WSASend。
     - Linux 使用 sendmmsg 或 sendmsg。
     - 其他平台默认使用 sendmsg。
   - 这种设计能针对不同系统的特点优化性能，而调用方完全不需要感知。
4. **代码复用**

- BufferList 提供了一致的接口，派生类只需实现特定功能。通过继承和多态机制，简化了不同发送策略的实现和维护。

实现了什么功能?

功能1：平台无关的统一接口

调用方通过 BufferList::Ptr buffer_list = BufferList::create(...) 获取实例，可以在任何系统上工作，不需要关心底层实现细节。

功能2：动态分配合适的派生类

根据 is_udp 和操作系统条件，动态选择最优的派生类，如:

- Linux 上的 sendmmsg 支持批量发送，性能优于逐一调用 sendto。
- Windows 的 WSASend 适用于非阻塞 I/O，效率更高。

功能3：降低复杂性

派生类的创建逻辑集中在 create 方法中，隐藏了复杂的条件判断和派生类细节，调用者无需重复实现这些逻辑。

总结

代码中的设计体现了工厂方法模式的核心思想：**延迟到子类决定创建哪种对象。**通过静态方法 BufferList::create:

1. **实现了解耦**：调用者不需要知道具体派生类，统一使用抽象接口 BufferList。
2. **提高了扩展性和平台适配能力**：新增派生类或优化现有策略时，只需修改 create，而调用方代码保持不变。
3. **优化了性能**：通过动态选择策略，在不同平台实现最佳性能。

这种设计不仅符合工厂模式的原则，还充分考虑了跨平台性能优化需求，体现了良好的架构思想。

工厂模式是一种创建对象的设计模式，其核心思想是将对象的创建过程封装起来，从而使客户端代码与具体的对象实现解耦。工厂模式主要分为以下几种形式:

1. 简单工厂模式 (Simple Factory Pattern)

定义

一个工厂类根据参数的不同，返回不同类型的实例。这是一种最简单的工厂模式实现。

特点

- 工厂类集中管理对象的创建。
- 客户端只需传递参数，不需要直接使用 new 创建对象。

例子

以一个饮料工厂为例，根据输入返回不同种类的饮料对象:

cpp
复制代码

```
class Beverage {
public:
```

```
    virtual void drink() = 0;
    virtual ~Beverage() = default;
};
class Coffee : public Beverage {
public:
    void drink() override {
        std::cout << "Drinking coffee.\n";
    }
};
class Tea : public Beverage {
public:
    void drink() override {
        std::cout << "Drinking tea.\n";
    }
};
class BeverageFactory {
public:
    static Beverage* createBeverage(const std::string& type) {
        if (type == "coffee") {
            return new Coffee();
        } else if (type == "tea") {
            return new Tea();
        } else {
            return nullptr;
        }
    }
};
// 使用示例
int main() {
    Beverage* drink = BeverageFactory::createBeverage("coffee");
    if (drink) {
        drink->drink();
        delete drink;
    }
    return 0;
}
```

优点

- 简单易用，适合简单对象创建。

缺点

- 工厂类职责过重，违反了**开闭原则**（每新增一种类型需要修改工厂类）。
- 不支持复杂对象结构。

2. 工厂方法模式 (Factory Method Pattern)

定义

定义一个创建对象的接口，但将具体实现延迟到子类。每个具体工厂负责一个具体产品的创建。

特点

- 遵循开闭原则。
- 每种产品对应一个工厂类。

例子

以操作系统日志记录器为例，不同操作系统有不同的实现：

cpp

复制代码

```
class Logger {
public:
    virtual void log(const std::string& message) = 0;
    virtual ~Logger() = default;
};
class WindowsLogger : public Logger {
public:
    void log(const std::string& message) override {
        std::cout << "Windows Logger: " << message << "\n";
    }
};
class LinuxLogger : public Logger {
public:
    void log(const std::string& message) override {
        std::cout << "Linux Logger: " << message << "\n";
    }
};
class LoggerFactory {
public:
    virtual Logger* createLogger() = 0;
    virtual ~LoggerFactory() = default;
};
class WindowsLoggerFactory : public LoggerFactory {
public:
    Logger* createLogger() override {
        return new WindowsLogger();
    }
};
class LinuxLoggerFactory : public LoggerFactory {
public:
    Logger* createLogger() override {
        return new LinuxLogger();
    }
};
// 使用示例
int main() {
    LoggerFactory* factory = new WindowsLoggerFactory();
    Logger* logger = factory->createLogger();
    logger->log("Hello, Factory Method!");
    delete logger;
    delete factory;
    return 0;
}
```

优点

- 解耦了产品的创建与使用。
- 易于扩展新产品类型。

缺点

- 增加了类的数量，复杂性提高。
- 每种新产品需要新增一个工厂类。

3. 抽象工厂模式 (Abstract Factory Pattern)

定义

提供一个接口，用于创建一系列相关或互相依赖的对象，而无需指定它们的具体类。

特点

- 适用于有多种产品族的场景。
- 工厂负责创建多个相关产品，而不仅仅是一个。

例子

以一个跨平台 GUI 工具包为例，支持创建按钮和窗口组件：

cpp
复制代码

```
class Button {
public:
    virtual void render() = 0;
    virtual ~Button() = default;
};
class Window {
public:
    virtual void render() = 0;
    virtual ~Window() = default;
};
class WindowsButton : public Button {
public:
    void render() override {
        std::cout << "Rendering Windows Button.\n";
    }
};
class LinuxButton : public Button {
public:
    void render() override {
        std::cout << "Rendering Linux Button.\n";
    }
};
class WindowsWindow : public Window {
public:
    void render() override {
        std::cout << "Rendering Windows Window.\n";
    }
};
class LinuxWindow : public Window {
public:
    void render() override {
        std::cout << "Rendering Linux Window.\n";
    }
};
```

```
class GUIFactory {
public:
    virtual Button* createButton() = 0;
    virtual Window* createWindow() = 0;
    virtual ~GUIFactory() = default;
};
class WindowsFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new WindowsButton();
    }
    Window* createWindow() override {
        return new WindowsWindow();
    }
};
class LinuxFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new LinuxButton();
    }
    Window* createWindow() override {
        return new LinuxWindow();
    }
};
// 使用示例
int main() {
    GUIFactory* factory = new WindowsFactory();
    Button* button = factory->createButton();
    Window* window = factory->createWindow();
    button->render();
    window->render();
    delete button;
    delete window;
    delete factory;
    return 0;
}
```

优点

- 提供了一个产品族的创建。
- 确保产品之间的兼容性。

缺点

- 扩展一个产品族时，需要修改抽象工厂接口，违反开闭原则。
- 实现复杂度较高。

4. 工厂模式在 ZLToolKit 中的应用

示例：BufferList::create

在 ZLToolKit 中，BufferList::create 是一个动态选择具体实现的工厂方法，既符合工厂方法模式的特点，又能针对不同条件优化对象创建。

特点

1. 动态选择合适的派生类：如 BufferSendTo、BufferSendMsg。

2. 隐藏了具体实现：用户通过统一的 BufferList 接口操作，无需关心派生类。
3. 易于扩展：如果未来增加新的发送策略，只需新增派生类并修改 create 方法。

优点

- **解耦**：调用方无需直接依赖派生类。
- **跨平台支持**：针对不同系统条件选择最优实现。
- **维护方便**：实现逻辑集中在 create 方法中，易于修改。

总结

工厂模式通过不同的形式（简单工厂、工厂方法、抽象工厂）实现了对象创建的封装和解耦，适合不同复杂度的场景。

在 ZLToolKit 中，工厂模式有效地提升了代码的可维护性、可扩展性和跨平台适配能力。