垃圾回收

(OOP)

刘知远 张正彦

liuzy@tsinghua.edu.cn

http://nlp.csai.tsinghua.edu.cn/~lzy/

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 语言集成查询LINQ解析器
- 迭代器模式
- 模板模式
- 策略模式

本讲内容提要

- 垃圾回收案例
- 单例模式
- 代理模式
- 适配器模式

垃圾回收

三种内存分配类型

- **静态 Static**
  - 指在编译时就能确定每个数据目标在运行时刻需要的存储空间需求
  - 比如C++类中的静态成员变量
- **栈式 Stack**
  - 在C/C++中，所有的方法调用都是通过栈来进行的，所有局部变量，形式参数都是从栈中分配内存空间的
- **堆式 Heap**
  - 堆由大片的可利用的块或空闲组成，堆中的内存可以按照任意顺序分配和释放
  - 在C/C++中，容易出现内存泄露，比如new之后忘记delete

垃圾回收

- 除智能指针之外，另一种管理内存的方式
- 垃圾回收（Garbage Collection，GC）是指一种自动的内存管理机制
  - 当某个程序占用的一部分内存空间不再被这个程序访问时，这个程序会借助垃圾回收算法向操作系统归还这部分内存空间
- 垃圾回收最早起源于LISP语言。当前许多语言如Java、C#、Smalltalk和D语言都支持垃圾回收器

垃圾回收：追踪

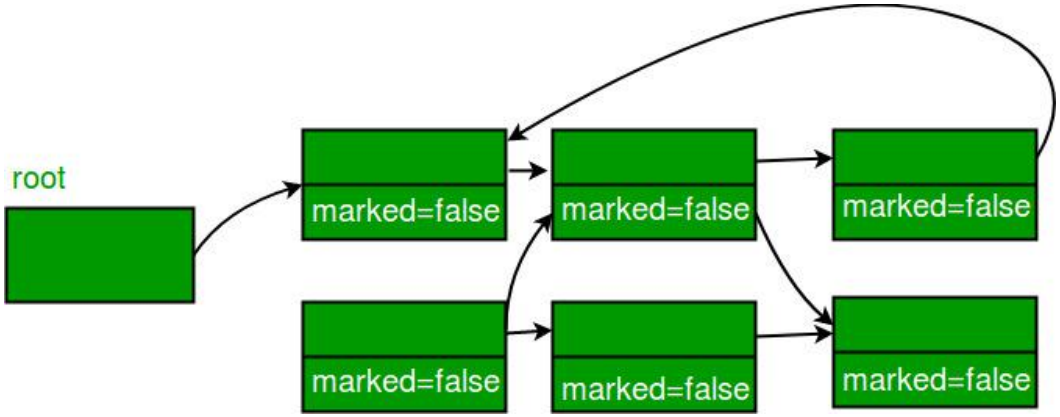
- **追踪是目前使用范围最广的技术**
  - 追踪算法从某些被称为 root 的对象开始，不断追踪可以被引用到的对象，这些对象被称为可到达的（reachable），其他剩余的对象就被称为 garbage，并且会被释放
- **可达对象主要有两类情况**
  - root对象，包括全局对象、调用栈（call stack）上的对象（包括内部变量与参数）
  - 从root对象开始，间接引用的对象，例如root对象的成员变量

Mark-and-Sweep

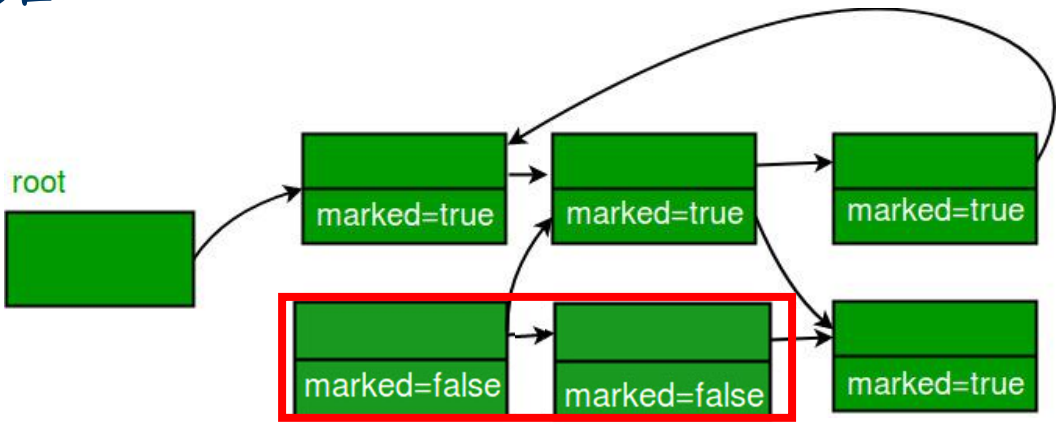
- 追踪算法中最经典的算法
- 第一步：Mark
  - 从root对象开始进行树遍历，每个访问的对象标注为“使用中”
- 第二步：Sweep
  - 扫描整个内存区域，对于标注为“使用中”的对象去掉该标志，对于没有该标注的对象直接回收掉

Mark-and-Sweep

- Mark开始前，所有对象都没有标志

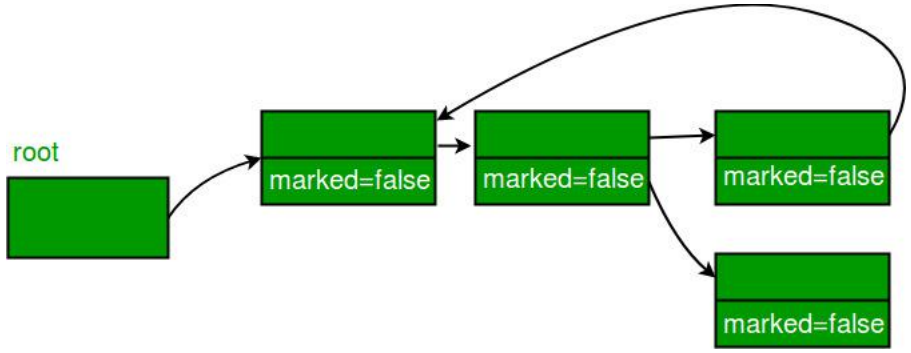


- 遍历之后



Mark-and-Sweep

- Sweep之后回收了无标注对象，并且清除了已有标注



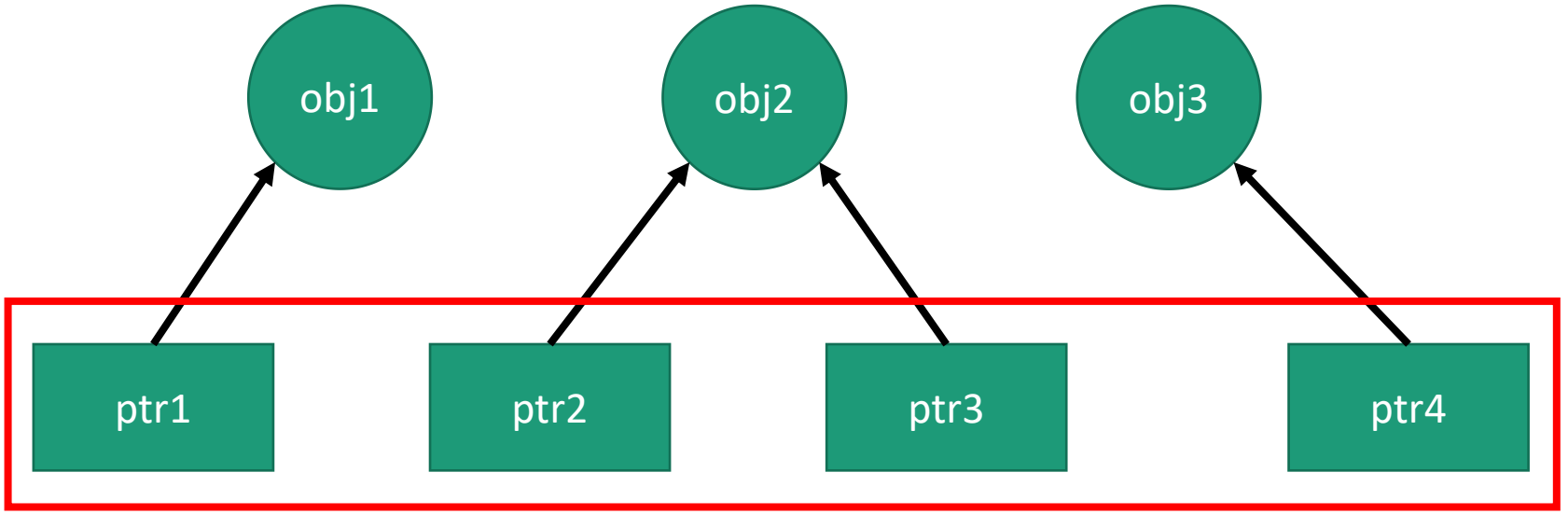
- 如何实现一个基于C++的Mark-and-Sweep机制？

Mark-and-Sweep实现思路

- **对于已有的对象，GC要进行全局的管理**
  - 记录对象引用情况的变化，如创建、赋值、销毁
  - 在内存不足时执行Mark-and-Sweep
  - 在程序结束时销毁所有对象
- **为了能够记录引用的变化，引入类似于智能指针的GC指针类管理对象的指针**
  - 所有对象的操作转为对于GC指针类的操作
  - 由GC指针类自己来更新引用的变化
  - 智能指针是自己存储关于自己的引用计数
  - GC指针可以修改和更新全局的引用情况

Mark-and-Sweep实现思路

- 对于类T的对象，创建之后不直接暴露给用户，而是返回一个指针类
  - 指针类：gc_ptr<T>
  - gc_ptr<T>.reference = new class T();
  - 任何T的操作都可以使用该指针类进行
  - 存在多个指针指向同一个对象



全局对象管理
(单例模式)

全局对象管理

- 设计一个全局对象管理器
  - GarbageCollection
  - 无论在程序哪里都可以对其进行操作
- 管理器的私有成员变量为一个GC对象的集合
  - set<gc_handle*, gc_handle_comparer>
- gc_handle是记录被管理对象信息的数据结构

```
struct gc_handle
{
    static const int counter_range = (int32_t)0x80000000;

    int counter = 0; // 对象的引用计数，有多少gc_ptr<T>指针指向它
    gc_record record; // 记录对象在堆上的开始地址以及对象的大小
    multiset<gc_handle*> references; // 记录该对象的成员变量的
    gc_handle
    bool mark = false; // 使用标记，在mark-and-sweep中被使用
};
```

全局对象管理

- 采用全局变量是一种最直接的方法：

```
set<gc_handle*, gc_handle_comparer> GarbageCollection = 0;  // 全局变量
// 加入一个新的被管理对象
void addHandle(gc_handle* handle) {  // 全局函数
    GarbageCollection.insert(handle);
}
// 查找某个对象是否被GC管理，根据 handle->record 中内存地址
gc_handle* findHandle(gc_handle* handle) {
    auto it = GarbageCollection.find(handle);
    return it == GarbageCollection.end() ? nullptr : *it;
}
```

- 这样的设计存在什么问题？

全局对象管理

- 用户可能访问到 GarbageCollection 并修改数据，不安全

```
set<gc_handle*, gc_handle_comparer> GarbageCollection;  // 全局变量

void addHandle(gc_handle* handle) {  // 全局函数
    GarbageCollection.insert(handle);
}

gc_handle* findHandle(gc_handle* handle) {
    auto it = GarbageCollection.find(handle);
    return it == GarbageCollection.end() ? nullptr : *it;
}
```

- 好的设计应当避免全局变量

静态成员

- 定义一个类，将数据封装为静态成员

```
class GarbageCollection {
private:
    //定义为私有的静态成员，使得用户无法直接访问
    static set<gc_handle*, gc_handle_comparer> gc_handles;
public:
    void addHandle(gc_handle* handle) {
         gc_handles.insert(handle);
    }
    gc_handle* findHandle(gc_handle* handle) {
        auto it = gc_handles.find(handle);
        return it == gc_handles.end() ? nullptr : *it;
    }
};
//实例化GarbageCollection来进行操作
GarbageCollection gc;
gc.addHandle(new gc_handle);
```

- 但在这样的实现中，每次需要实例化GarbageCollection再进行操作，效率不高

静态方法

- 定义一个类，将函数实现为静态方法：

```
class GarbageCollection {
private:
    //定义为私有的静态成员，使得用户无法直接访问
    static set<gc_handle*, gc_handle_comparer> gc_handles;
public:
    static void addHandle(gc_handle* handle) {
         gc_handles.insert(handle);
    }
    static gc_handle* findHandle(gc_handle* handle) {
        auto it = gc_handles.find(handle);
        return it == gc_handles.end() ? nullptr : *it;
    }
};
//定义为静态方法，无需实例化就可以进行操作
GarbageCollection::addHandle(new gc_handle);
```

- 这样的实现克服了之前的问题，但是否完美?

静态方法 & 虚函数

- 如果我们有多种不同的GarbageCollection……

```
class BaseGC {
public:
    static virtual void addHandle(gc_handle* handle) = 0;
    static virtual gc_handle* findHandle(gc_handle* handle) = 0;
};

class SimpleGC : public BaseGC {
    static set<gc_handle*, gc_handle_comparer> gc_handles;
public:
    static void addHandle(gc_handle* handle) { ... }
    static gc_handle* findHandle(gc_handle* handle) { ... }
};

class NotSimpleGC : public BaseGC { ... };
```

- 可以这么写吗?

静态方法 + 虚函数 = 编译错误

- 静态方法+虚函数的问题
  - 静态方法不可以是虚的!
- 我们使用静态方法的根本目的:
  - 无论在何处调用，都会访问到相同的函数和变量
  - 无需多次实例化以提升工作效率
- 能否不使用静态方法达到我们的目的?

单例模式

- 属于创建型模式
- 在单例模式中
  - 单例类只能有一个实例
  - 单例类必须自己创建自己的唯一实例
  - 单例类必须给所有其他对象提供这一实例

- 实现单例模式主要关键点
  - 构造函数不对外开放，一般为Private
  - 通过一个静态方法或者枚举返回单例类对象

单例模式

- 所谓单例，就是只能构造一份实例的类

```
class GarbageCollection {
    // 显式删除拷贝构造函数与赋值操作符，防止出现多份实例
    GarbageCollection(const GarbageCollection &) = delete;
    void operator =(const GarbageCollection &) = delete;

    set<gc_handle*, gc_handle_comparer> gc_handles;
    GarbageCollection() { }  // 将构造函数设为私有，防止被调用

    static GarbageCollection _instance;  // 全局唯一的实例
public:
    // 只允许通过调用这一方法获取GarbageCollection类的实例
    static GarbageCollection &instance() {
        return _instance;
    }
    // addHandle和findHandle实现为成员函数而非静态方法
    void addHandle(gc_handle* handle) { ... }
    gc_handle* findHandle(gc_handle* handle) { ... }
};
```

单例模式

- 调用单例

```
// GarbageCollection类的定义
class GarbageCollection { ... };
// 定义类中的静态成员，该单例的唯一实例在此被构造
GarbageCollection GarbageCollection::_instance;

int main() {
    // 由于删去了拷贝构造函数，必须存为引用
    GarbageCollection &gc = GarbageCollection::instance();
    gc.addHandle(new gc_handle);
}
```

- 单例模式封装了全局性的变量，无需多次实例化，无需依靠静态访问方法，很好的满足了我们的需要

惰性初始化（Lazy Initialization）

- 能否让单例模式在使用时自动构造单例实例？

```
class GarbageCollection {
    // ...
public:
    static GarbageCollection &instance() {
        //这个GarbageCollection只有在第一次访问instance()时才会被实例化
        static GarbageCollection _instance;
        return _instance;
    }
    // ...
};
```

- 函数内的静态变量在程序第一次执行到其定义时才会被构造
- 因此，第一次访问instance()会实例化单例，之后的访问会直接返回已有的单例

单例模式：陷阱！

- 刚才的实现真的是没问题的吗？
- 单例需要避免的情况：
  - 实例被重复构造
    - 由于构造函数为private，且拷贝构造函数、赋值操作符被显式删除，故无法重复构造。
  - 实例被意外删除
    - 由于返回值必须以引用形式存储（而非指针或实例），故无法被意外删除。
- 当前的实现似乎完美规避了重复构造与重复删除两种情况

事实上，当前实现无法避免意外删除！

单例模式：陷阱!

- 考虑下面的代码，可以将单例意外删除

```
GarbageCollection &gc = GarbageCollection::instance();
delete &gc;     // 可以成功执行
gc.addHandle(new gc_handle); // 运行时错误!
```

- 所以应当把析构函数也设为私有！可以规避上述操作

```
class GarbageCollection {
private:
    ~GarbageCollection() { ... }
// ...
};

/* 编译信息:
 *   error: calling a private destructor of class
'GarbageCollection'
 *       delete &gc;
 *              ^
 */
```

单例模式 + 虚函数

- 使用虚函数实现单例的继承:

```
class BaseGC {
public:  // 定义为虚成员方法
    virtual void addHandle(gc_handle* handle) = 0;
    virtual gc_handle* findHandle(gc_handle* handle) = 0;
};

class SimpleGC : public BaseGC {
    // ...单例相关的一大堆逻辑...
    set<gc_handle*, gc_handle_comparer> gc_handles;
    SimpleGC() { }
public:
    virtual void addHandle(gc_handle* handle) { ... }
    virtual gc_handle* findHandle(gc_handle* handle) { ... }
};

class NotSimpleGC : public BaseGC { ... };
// ... 接下一页 ...
```

单例模式 + 虚函数

```
// ... 接上一页 ...
void doStuff(BaseGC *gc) {
    gc->addHandle(new gc_handle);
}

int main() {
    doStuff(&SimpleGC::instance());      // 1
    doStuff(&NotSimpleGC::instance());
    doStuff(&SimpleGC::instance());      // 2
}
```

- 唯一的不便：单例相关的逻辑较多，需要在派生类中分别实现单例代码，尤其是每个单例派生类的 `instance()`
- 能否有更简单的实现来复用代码呢?

奇特的递归模板模式

- “奇特的递归模板模式”：CRTP
  Curiously Recurring Template Pattern

```
template <class Derived>  // 模板参数为派生类类型
class Singleton {
    Singleton(const Singleton &) = delete;
    void operator =(const Singleton &) = delete;
protected:
    Singleton() {}
    virtual ~Singleton() {}
public:
    static Derived &instance() {  // 魔法在此发生，Derived为Singleton
的派生类类型，通过模板传入，并在基类中创建唯一实例
        static Derived _instance;
        return _instance;
    }
};
class SimpleGC : public BaseGC,
                 public Singleton<SimpleGC> {
    friend class Singleton<SimpleGC>; // ...
}
```

CRTP + 多重继承

- 基于Singleton类实现GC派生类：

```
// 下面的继承声明是合法的，因为Singleton基类中不包含派生类的实例
class SimpleGC : public BaseGC,
                 public Singleton<SimpleGC> {
    // 友元声明是必要的，因为Singleton类需要访问派生类的构造函数，
    // 而为了实现单例，构造函数是私有的
    friend class Singleton<SimpleGC>;
    // ... 只需实现GC逻辑即可
}
```

- 注意，不能直接将Singleton类的逻辑实现在BaseGC类中，单例功能与其他功能剥离
  - 否则BaseGC将成为模板类，无法脱离模板参数存在

关于CRTP

- **CRTP是实现多态的另一种方式（不局限在单例模式之中使用）**
  - 利用C++模板，让编译器生成重复代码
- **与虚函数不同，实现的是编译期多态**
  - 需要在编译期确定实际被调用的函数
- **考虑一个任意派生类实例为参数的函数，在函数中调用实例的方法：**
  - 使用虚函数实现：运行时通过虚函数表寻找调用的方法
  - 使用CRTP实现：函数需要被实现为模板函数，编译时由编译器为每种被调用的派生类进行模板实例化

关于单例模式

- 单例模式是存在争议的一种设计模式
- 优点：
  - 实现似乎比较简单
  - 以相对安全的形式提供可供全局访问的数据
- 缺点：
  - 难以完全正确地实现，安全隐患在各种特殊情况下可能仍然存在，防不胜防
  - 违反了面向对象单一职责原则
  - 滥用这一方法会使得实际的依赖关系变得隐蔽

多选题 1分

设置

以下关于单例模式说法不正确的是

A 静态函数不可以是虚函数

B 我们可以使用惰性初始化，返回单例实例的引用

C 单例模式需要将析构函数设置为私有，以避免实例被意外删除

D CRPT模式实现了运行时多态

提交

GC单例的常用接口

- 几个重要的操作
  - 创建一个对象后，将其加到GC中
  - 根据给定的对象地址找到其gc_handle
  - gc_ptr<T>指针创建时，对象和指针绑定
  - gc_ptr<T>指针析构时，对象和指针解绑
  - gc_ptr<T>指针内容被修改，指针解绑旧对象，绑定新对象
- 对象和指针绑定是有两种情况
  - 指针本身是全局变量或者栈上的变量，那么此对象在此时可以看做一个root对象，引用数counter增加
  - 指针是某对象的成员变量，那么增加两对象gc_handle之间的引用
- 对于解绑处理的情况和绑定正好相反

GC中添加新对象

- 创建一个对象后，将其加到GC中

```
void GarbageCollection::gc_alloc(gc_record record) // 传入record
{
    auto handle = new gc_handle; // 创建一个新的handle
    handle->record = record;
    handle->counter = 1; // 开始的时候计数为1
    vector<gc_handle*> garbages;
    { // 检查是否需要进行GC，如果当前堆上内存不够则需要进行GC
      // GC操作时禁止其它的修改，因此引入的锁机制
        lock_guard<mutex> guard(gc_lock); // 加锁
        this->gc_handles.insert(handle); // 加入新的handle
        gc_current_size += handle->record.length; // 更新当前堆大小
        if (gc_current_size > gc_max_size) // 如果超过最大大小，回收
            gc_force_collect_unsafe(garbages); // 把找到的garbage放到
        garbages中，这里传的是garbages的引用
    }
    gc_destroy_unsafe(garbages); // 销毁garbage，如果是空就不操作
}
```

GC中查找对象

- 根据给定的对象地址找到其gc_handle
  - void* handle 为对象的地址
  - gc_handle_dummy只需要一个对象的地址就可以创建一个gc_handle来进行查找

```
gc_handle* GarbageCollection::gc_find_unsafe(void* handle)
{
    gc_handle_dummy dummy;
    dummy.record.start = handle;
    gc_handle* input = reinterpret_cast<gc_handle*>(&dummy);
    auto it = this->gc_handles.find(input); // stl中set的查找功能
    return it == this->gc_handles.end() ? nullptr : *it;
}
```

GC中绑定指针和对象

- 创建一个新的gc_ptr<T>指针时，更新追踪信息
  - handle_reference是新gc_ptr<T>指针的地址
  - handle是堆上对象的地址
  - 调用了gc_ref_connect_unsafe将gc_ptr<T>指针与堆上对象绑定

```
void GarbageCollection::gc_ref_alloc(void** handle_reference, void*
handle)
{
    lock_guard<mutex> guard(gc_lock); // 加锁
    gc_ref_connect_unsafe(handle_reference, handle);
}
```

GC中绑定指针和对象

- gc_ptr<T>指针与堆上对象绑定

```
void GarbageCollection::gc_ref_connect_unsafe(void** handle_reference,
void* handle)
{
    gc_handle* parent = nullptr;
    if (auto target = gc_find_unsafe(handle)) // 确定该对象是否由GC管理
    {
    if (parent = gc_find_parent_unsafe(handle_reference))
    { // 查找该指针是否为某个GC管理对象的成员，如果是构建引用结构
        parent->references.insert(target);
    }
    else
    { // 不是已有对象的成员，说明该对象是一个root对象，引用数增加
        target->counter++;
    }
    }
}
```

GC中解绑指针和对象

- gc_ptr<T>指针析构时，数据和指针解绑，更新追踪信息
  - 调用了gc_ref_disconnect_unsafe函数

```
void GarbageCollection::gc_ref_dealloc(void** handle_reference, void*
handle)
{
    lock_guard<mutex> guard(gc_lock);
    gc_ref_disconnect_unsafe(handle_reference, handle);
}
```

GC中解绑指针和对象

- gc_ptr<T>指针与堆上对象解绑

```
void GarbageCollection::gc_ref_disconnect_unsafe(void** handle_reference,
void* handle)
{
    gc_handle* parent = nullptr;
    if (auto target = gc_find_unsafe(handle))
    {
    if (parent = gc_find_parent_unsafe(handle_reference))
    { // 删除对象和对象成员之间的引用关系
        parent->references.erase(target);
    }
    else
    { // root对象，引用数减少
        target->counter--;
    }
    }
}
```

GC中修改指针指向对象

- 修改gc_ptr<T>指针指向的堆上对象
  - handle_reference指针的地址
  - old_handle原对象地址
  - new_handle新对象地址
  - 先与旧的对象解绑，然后绑定新的对象

```
void GarbageCollection::gc_ref(void** handle_reference, void* old_handle,
void* new_handle)
{
    lock_guard<mutex> guard(gc_lock);
    gc_ref_disconnect_unsafe(handle_reference, old_handle, false);
    gc_ref_connect_unsafe(handle_reference, new_handle);
}
```

GC指针
(代理模式)

GC指针

- gc_ptr<T>
- 能够包裹指针，具有指针的各项功能，并能够进行GC管理，在发生拷贝、赋值、创建、析构等操作时，进行GC的追踪
- 需要考虑各种会影响引用情况的操作
  - 默认构造函数
  - 拷贝构造函数
  - 移动构造函数
  - 赋值运算符
  - 类型转换
  - 析构函数

代理/委托

- 在一些应用中，直接访问对象往往会带来诸多问题
  - 要访问的对象在远程的机器上。
  - 被访问对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问
  - 直接访问会给使用者或者系统结构带来不必要的麻烦
  - 被访问对象需要实时根据访问者的行为做出诸多复杂处理
- 我们可以在被访问对象上加上一个访问层，将复杂操作包裹在内部不对外部类开放，仅对外开放功能接口，即可完成上述要求，这就是代理/委托模式

具有指针的功能

- 重载->运算符
- 将T类的指针作为私有成员变量reference

```
template<typename T>
class gc_ptr
{
    private:
    // ...
        T* reference = nullptr;
    public:
    // ...
        T* operator->()
        {
            return reference;
        }
}
```

默认构造函数

- 调用gc_ref_alloc函数
  - 将当前GC指针的地址this和空指针进行绑定
  - 代表当前没有指向任何对象
  - 由于是空指针，因此不会进行操作

```
gc_ptr()
{
    GarbageCollection::instance().gc_ref_alloc((void**)this, nullptr);
}
```

拷贝构造函数

- 调用gc_ref_alloc函数
  - 拷贝堆上对象的指针
  - 将新的GC指针与堆上对象绑定
  - 这里用handle_of获得对象在堆上真正的起始地址

```
gc_ptr(const gc_ptr<T>& ptr) :reference(ptr.reference) // 拷贝指针
{
    GarbageCollection::instance().gc_ref_alloc((void**)this,handle_of(ref
erence)); // 绑定指针和对象
}
```

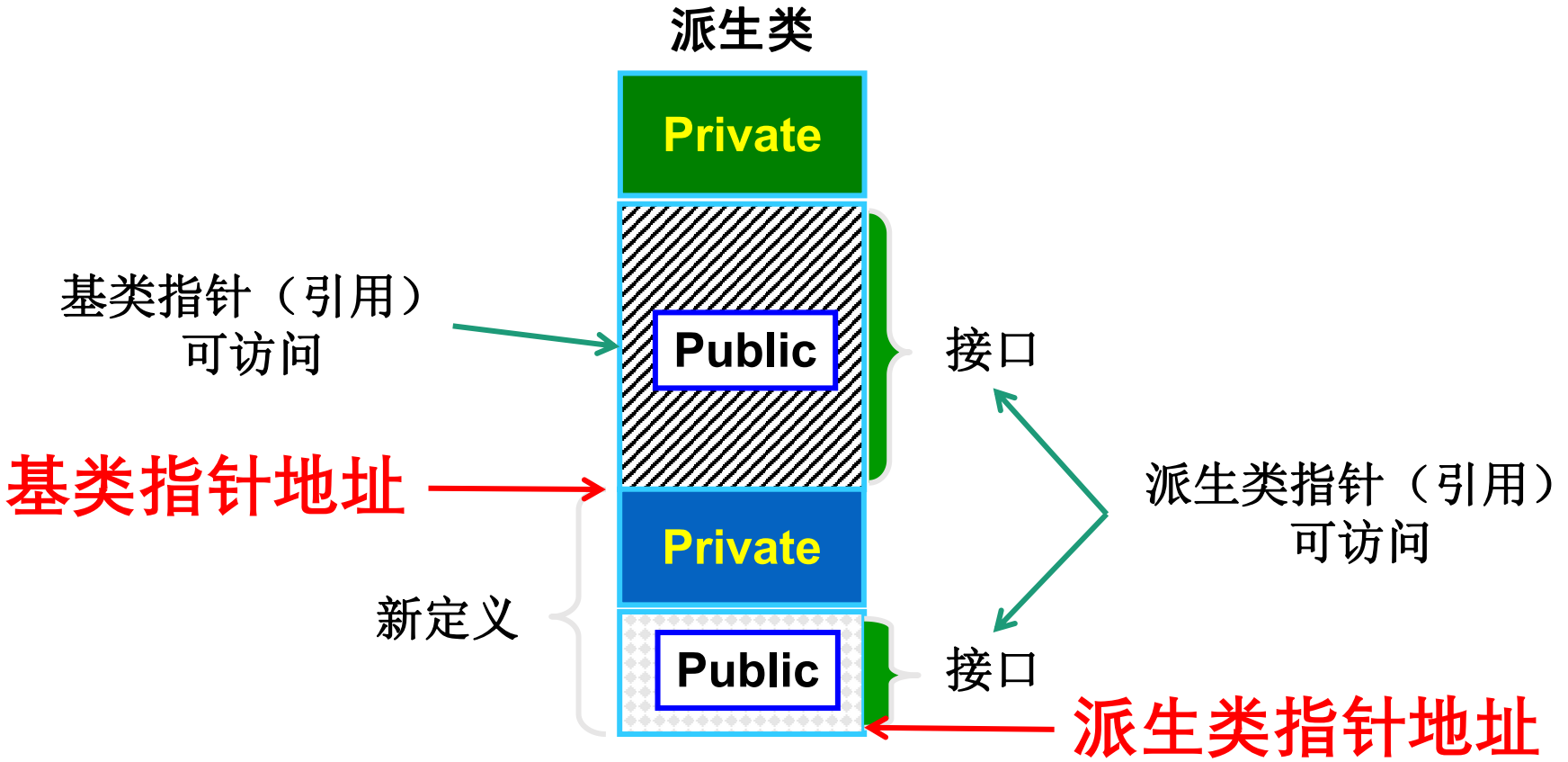
```
void* handle_of(T* reference)
{
    return reference ? static_cast<enable_gc*>(reference)->record.start :
nullptr; // 获得真正地址
}
```

真正的起始地址

- 为什么会有真正的起始地址这个概念
  - enable_gc是一个虚基类，提供了GC的基本信息
  - 假设A类继承了enable_gc，B类继承了A类
  - 对于一个B类的对象x，可以被gc_ptr<B>指向
  - 也可以被gc_ptr<A>指向，这里发生向上类型转换
  - C++里指针发生向上类型转换的时候地址也会变化
  - gc_ptr<A>中的reference就不再是真正的起始地址

指针（引用）的向上转换

- 当派生类的指针（引用）被转换为基类指针（引用）时，不会创建新的对象，但只保留基类的接口。



移动构造函数

- 首先调用gc_ref_alloc函数
  - 新的GC指针和堆上对象绑定
- 再调用gc_ref函数
  - 原GC指针需要置为空

```
gc_ptr(gc_ptr<T>&& ptr) :reference(ptr.reference)
{
    GarbageCollection::instance().gc_ref_alloc((void**)this,
handle_of(reference));
    ptr.reference = nullptr;
    GarbageCollection::instance().gc_ref((void**)&ptr,
handle_of(reference), nullptr);
}
```

赋值运算符

- 该情况下需要把原来的对象指针替换为新的指针
  - 调用gc_ref
  - 先取消对原来old_handle的依赖
  - 再建立对于new_handle的依赖

```
gc_ptr<T>& operator=(const gc_ptr<T>& ptr)
{
    void* old_handle = handle_of(reference);
    reference = ptr.reference;
    void* new_handle = handle_of(reference);
    GarbageCollection::instance().gc_ref((void**)this, old_handle,
    new_handle);
    return *this;
}
```

类型转换

- GC指针指向的不是T类型，因此需要类型转换
  - 该实现基本与拷贝构造类似
  - reference(ptr.reference)完成了指针类型转换

```
template<typename U>
gc_ptr(const gc_ptr<U>& ptr):reference(ptr.reference) // 隐式转换
{
    GarbageCollection::instance().gc_ref_alloc((void**)this,
handle_of(reference));
}
```

析构函数

- 调用gc_ref_dealloc取消GC指针和对象地址的依赖

```
~gc_ptr()
{
    GarbageCollection::instance().gc_ref_dealloc((void**)this,
handle_of(reference));
}
```

“变”与“不变”

- gc_ptr<T>与 T* 有相同的接口
  - 操作符：*和->
  - 赋值操作符与初始化（拷贝构造）
  - 释放（析构）
- gc_ptr<T>比 T* 增加了一些控制操作
- 常被称为“代理”模式
  - 接口不变，功能变化
  - 用于对被代理对象进行控制，如引用计数控制、权限控制、远程代理、延迟初始化等等
  - 代理类就好比被代理类的“经纪人”，一方面提供被代理类所有接口的功能，另一方面可以同时进行额外的控制操作。

垃圾回收

Mark-and-Sweep

- 首先把root对象给找出来

```
void GarbageCollection::gc_force_collect_unsafe(vector<gc_handle*>&
garbages)
{
    vector<gc_handle*> markings;

    for (auto handle : this->gc_handles)
    {
        if (handle->mark = handle->counter > 0) // root对象的counter大于0
        { // if语句中同时完成了mark的设定，不是root则为FALSE
            markings.push_back(handle); // 加入到markings中
        }
    }
    // ...下一页...
```

Mark-and-Sweep

- 根据 markings 进一步进行进行BFS

```
void GarbageCollection::gc_force_collect_unsafe(vector<gc_handle*>&
garbages)
{
    // ...接上页...
    for (int i = 0; i < (int)markings.size(); i++)
    {
        auto ref = markings[i];
        for (auto child : ref->references)
        {
            if (!child->mark) // 把root的间接引用对象加入数组，并标记
            {
                child->mark = true;
                markings.push_back(child); // 这会改变markings的size
                // 因此新加入的元素也会完成对他们references的遍历
            }
        }
    }
    // ...下一页...
```

Mark-and-Sweep

- Sweep

```
void GarbageCollection::gc_force_collect_unsafe(vector<gc_handle*>&
garbages)
{
    // ...接上页...
    for (auto it = this->gc_handles.begin(); it != this-
    >gc_handles.end();)
    {
        if (!(*it)->mark)
        {
            auto it2 = it++; // it指向下一个元素，it2存下了it的原值
            garbages.push_back(*it2); // 将需要删除的gc_handle放入garbages
            this->gc_handles.erase(it2); // 并从gc_handles移除
        }
        else
        {
            it++;
        }
    }
}
```

Mark-and-Sweep

- 上述函数完成了对于向量引用的更新
  - vector<gc_handle*>& garbages
- 之后调用gc_destroy_unsafe完成垃圾的析构

```
void GarbageCollection::gc_destroy_unsafe(vector<gc_handle*>& garbages)
{
    for (auto handle : garbages)
    {
        gc_destroy_unsafe(handle);
    }
    gc_last_current_size = gc_current_size;
}
```

Mark-and-Sweep

- 上述函数完成了对于向量引用的更新
  - vector<gc_handle*>& garbages
- 之后调用gc_destroy_unsafe完成垃圾的析构
  - record.handle为一个enable_gc的指针，通过调用析构函数完成垃圾的回收

```
void GarbageCollection::gc_destroy_unsafe(gc_handle* handle)
{
    handle->record.handle->~enable_gc();
    gc_current_size -= handle->record.length; // 计算析构后堆上的数据规模
    free(handle->record.start);
    delete handle; // 删除控制句柄
}
```

虚析构函数

- 使用enable_gc的指针完成对于堆上对象的析构

```
class enable_gc
{
    ...
    public:
        enable_gc();
        virtual ~enable_gc();
};
```

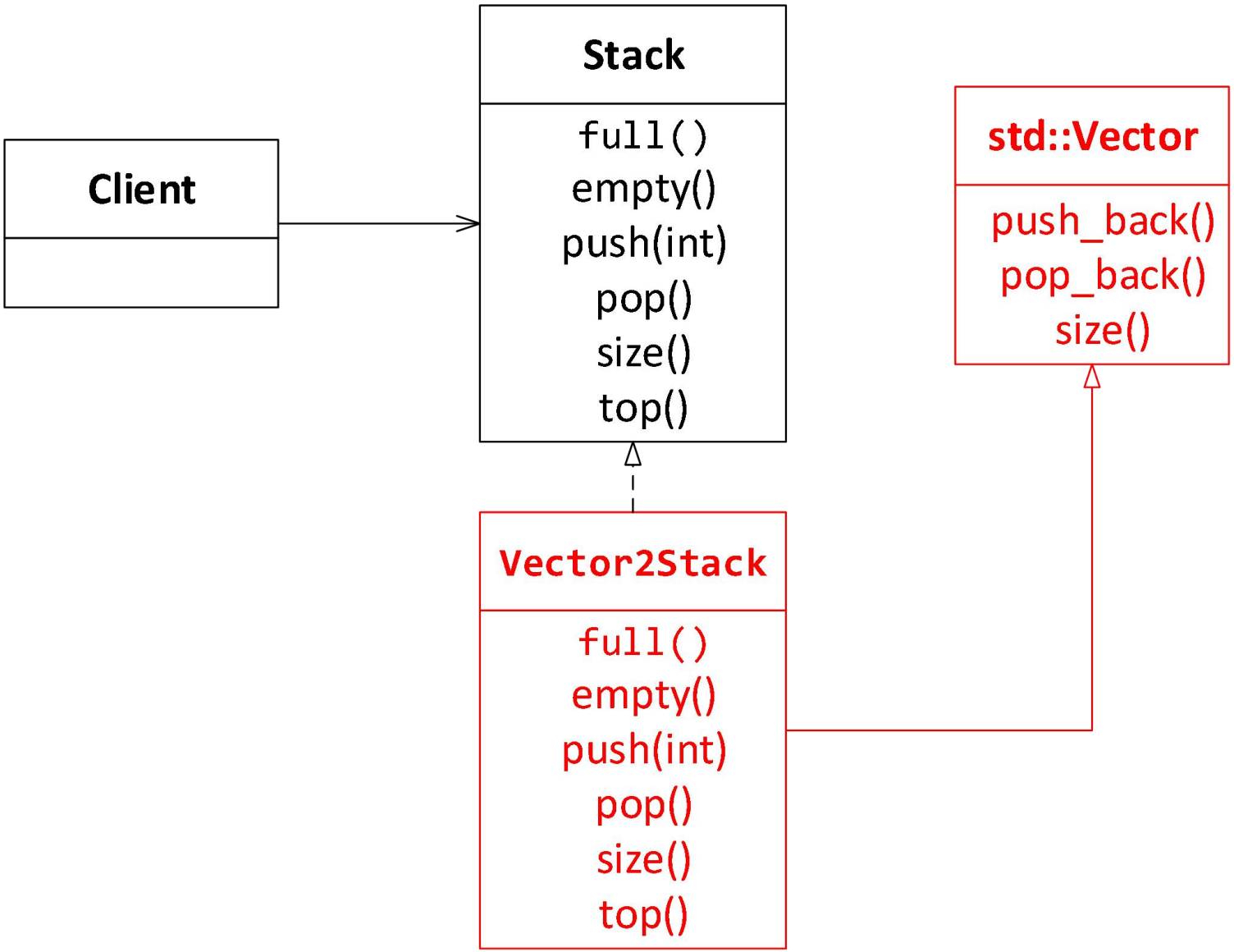
```
class A : public enable_gc
{
    public:
    A(int a) {}
    ~A() { }
};
```

适配器

适配器

- 类似于代理模式，也是对于一个类进行二次包装
- 概述
  - 适配器模式将一个类的接口转换成客户希望的另一个接口，从而使得原本由于接口不兼容而不能一起工作的类可以在统一的接口环境下工作。
- 结构
  - 目标（Target）：客户所期待的接口。
  - 需要适配的类（Adaptee）：需要适配的类。
  - 适配器（Adapter）：通过包装一个需要适配的类，把原接口转换成目标接口。

使用Vector实现Stack



适配器——实现

```
//直接继承vector并改造接口，采用私有继承可以使得外界只能接触到Vector2Stack中的接口

class Vector2Stack : private std::vector<int>, public Stack {
public:
    Vector2Stack(int size) : vector<int>(size) { }
    bool full() { return false; }
    bool empty() { return vector<int>::empty(); }
    void push(int i) { push_back(i); }
    void pop() { pop_back(); }
    int size() { return vector<int>::size(); }
    int top() { return back(); }
};
```

适配器——实现

```
int main(int argc, char *argv[]) {

    Vector2Stack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        std::cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

```
4
3
2
1
```

适配器 与 代理/委托

- **相似：**
  - 均是在被访问对象之上进行封装
  - 均提供被封装对象的功能接口供外部使用

- **不同：**
  - 代理不会改变接口，但适配器可能会
  - 代理不会改变功能，但适配器可能会
  - 适配器不会增加控制，代理可能会
  - 适配器的核心要素是变换接口，代理的核心要素是分割访问对象与被访问对象以减少耦合，并能在中间增加各种控制功能。

单选题 1分

设置

下列代码中，compForSingle类实现了比较两个数a与b大小的功能，而compForDouble则实现了比较(a1,a2)与(b1,b2)大小的功能，请问在compForDouble的实现过程中，使用了哪种设计模式

```
class compForSingle {
public:
	int calc(int a, int b) {
		if (a < b) return -1;
		if (a > b) return 1;
		return 0;
	}
};

class compForDouble : public compForSingle {
public:
	int calc(int a1, int a2, int b1, int b2) {
		int res1 = compForSingle::calc(a1, b1);
		int res2 = compForSingle::calc(a2, b2);
		if (res1 == 1) return 1;
		if (res1 == -1) return -1;
		if (res2 == 1) return 1;
		if (res2 == -1) return -1;
		return 0;
	}
};
```

A 适配器模式

B 代理模式

C 策略模式

D 模板方法

提交

本节课

- 介绍了垃圾回收的基本原理
- 使用单例模式和代理模式完成了C++垃圾回收机制的实现，也涉及与代理模式类似的适配器模式
- 本节课案例来自于轮子哥vczh的github项目
  - https://github.com/vczh/vczh_toys/tree/master/CppGarbageCollection
- 拓展阅读
  - 《设计模式》3.5(单件)，4.1(适配器)，4.7(代理)

结束