

cly

博客园 首页 新随笔 联系 订阅 管理 50 Posts :: 2 Stories :: 26 Comments :: 0 Trackbacks

公告

昵称: 戒色
园龄: 7年10个月
粉丝: 51
关注: 0
[+加关注](#)

搜索

常用链接

[我的随笔](#)
[我的评论](#)
[我的参与](#)
[最新评论](#)
[我的标签](#)

我的标签

[设计模式\(17\)](#)
[算法 排序\(4\)](#)
[引用\(1\)](#)
[执行顺序\(1\)](#)
[算法\(1\)](#)
[算法 查找\(1\)](#)
[abstractFactory抽象工厂\(1\)](#)
[Adapter适配器模式\(1\)](#)
[Bridge桥接模式\(1\)](#)
[Builder建造者模式\(1\)](#)
[更多](#)

随笔分类(55)

[面向对象\(29\)](#)

C++设计模式-Bridge桥接模式

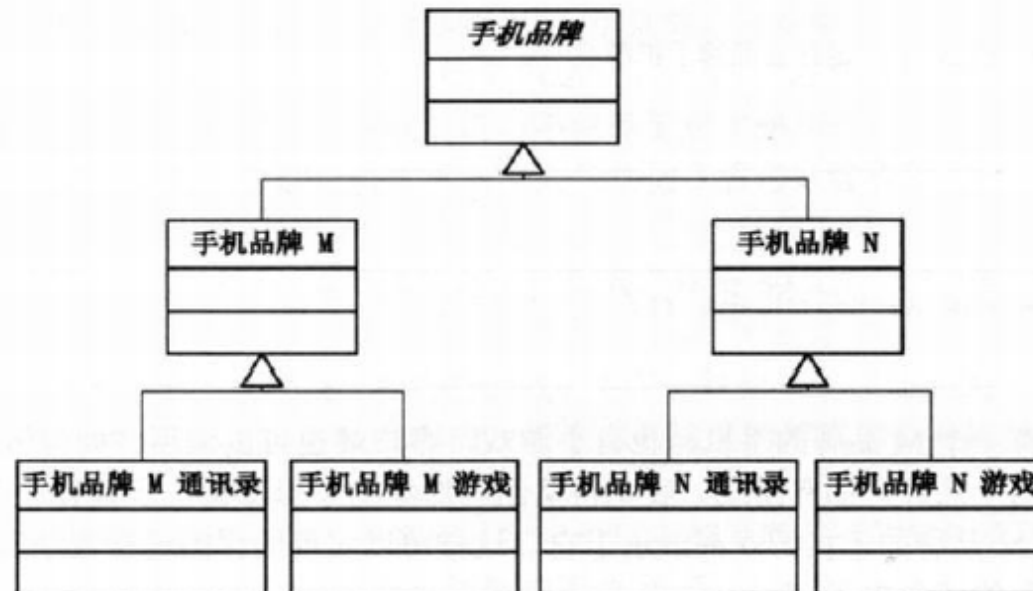
作用：将抽象部份与它的实现部份分离，使它们都可以独立地变化。

将抽象(Abstraction)与实现(Implementation)分离，使得二者可以独立地变化。

桥接模式号称设计模式中最难理解的模式之一，关键就是这个抽象和实现的分离非常让人奇怪，大部分人刚看到这个定义的时候都会认为实现就是继承自抽象，那怎么可能将他们分离呢。

《大话设计模式》中就Bridge模式的解释：

手机品牌和软件是两个概念，不同的软件可以在不同的手机上，不同的手机可以有相同的软件，两者都具有很大的变动性。如果我们单独以手机品牌或手机软件为基类来进行继承扩展的话，无疑会使类的数目剧增并且耦合性很高，（如果更改品牌或增加软件都会增加很多的变动）两种方式的结构如下：



设计模式(19)
算法, 数据结构(7)

随笔档案(50)

2013年9月 (1)
2013年7月 (19)
2013年6月 (8)
2012年11月 (1)
2012年7月 (7)
2012年6月 (8)
2011年7月 (1)
2011年5月 (5)

最新评论

1. Re:C++设计模式-Composite组合模式

感谢, 但是有个问题组合对象删除时, this-

>m_ComVec.erase(&com);, erase里面的参数应该是迭代器, 可以用
vector::iterator iter = find(this->.....
--MegamindLS

2. Re:C++设计模式-Flyweight享元模式

UnsharedConcreteFlyweight
这个类怎么没有用到呢

--WUZX

3. Re:C++设计模式-Factory工厂模式

受益匪浅。学到不少东西。

--最最最爱小盖儿

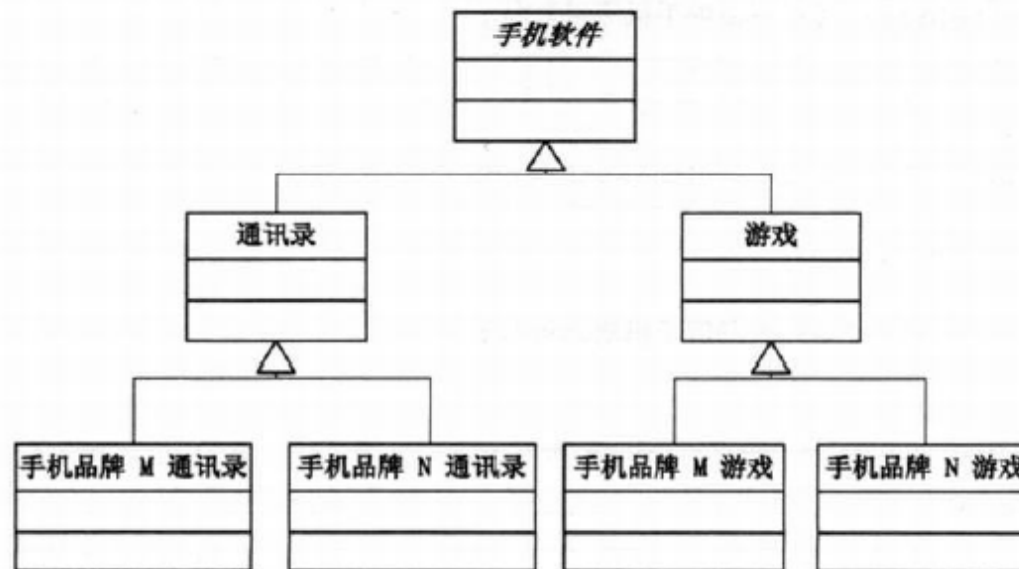
4. Re:C++设计模式-State状态模式

@蓝色心情2010博主贴的代码 内存泄漏了! 示例代码给人的感觉不够形象和直接! ...

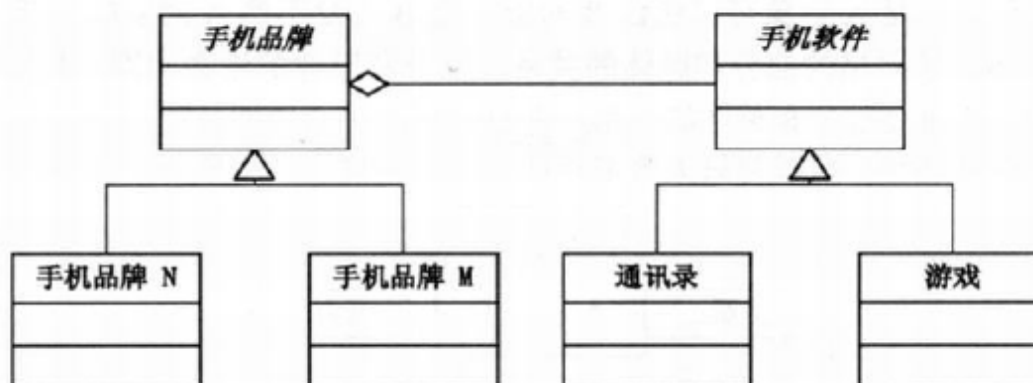
--AllKillMan

5. Re:C++设计模式-Observer观察者模式

@wenwenxiong应该是没仔细看吧, Observer子类对象的指针并不



所以将两者抽象出来两个基类分别是PhoneBrand和PhoneSoft, 那么在品牌类中聚合一个软件对象的基类将解决软件和手机扩展混乱的问题, 这样两者的扩展就相对灵活, 剪短了两者的必要联系, 结构图如下:



这样扩展品牌和软件就相对灵活独立, 达到解耦的目的!

UML结构图如下:

是在使用过程中attach和detach动态创建的! 而是在main函数中创建的。attach和detach中只是传参数进去的.....

--AllKillMan

阅读排行榜

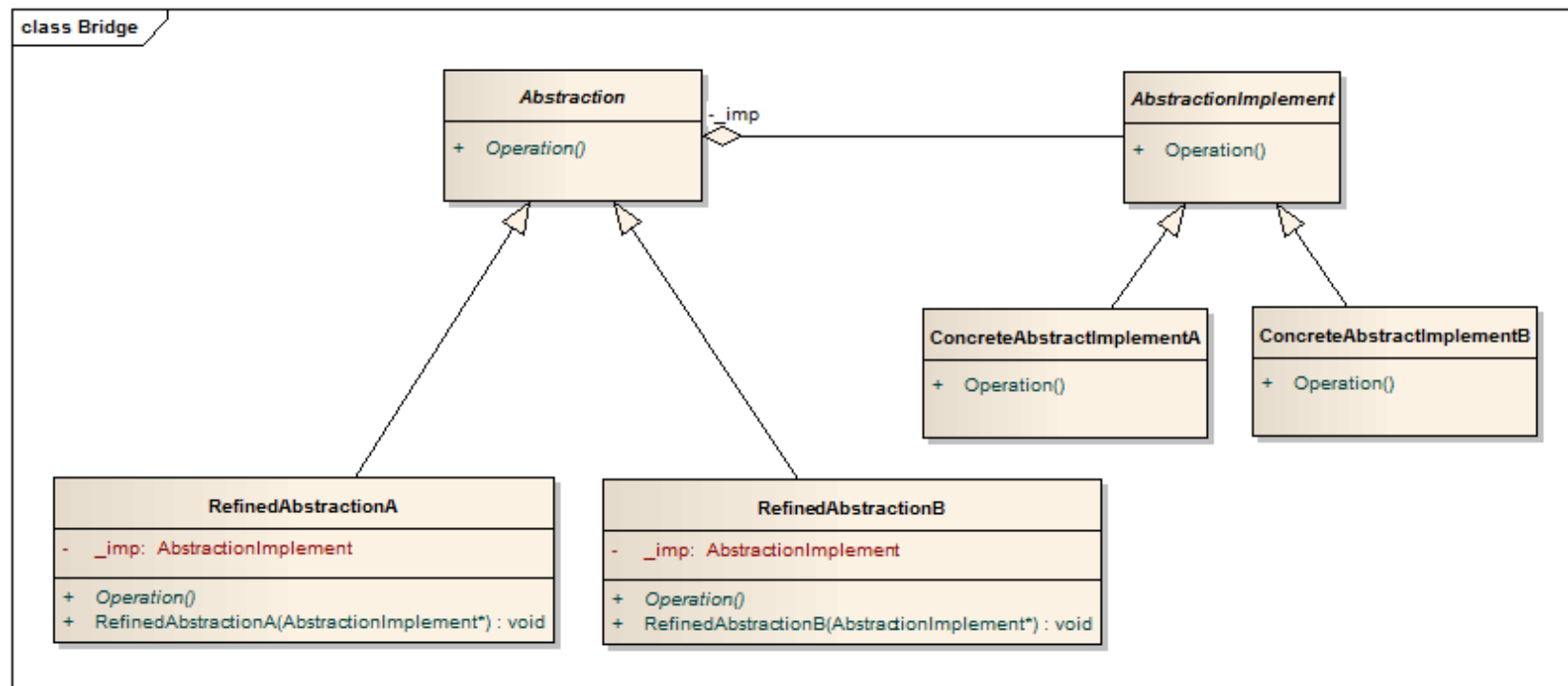
1. C++设计模式-Observer观察者模式(16274)
2. C++设计模式-Factory工厂模式(11644)
3. C++设计模式-Singleton(10423)
4. C++设计模式-Adapter适配器模式(10194)
5. C++设计模式-Bridge桥接模式(9493)

评论排行榜

1. C++设计模式-Observer观察者模式(7)
2. C++设计模式-Singleton(4)
3. C++设计模式-Factory工厂模式(3)
4. C++设计模式-State状态模式(3)
5. C++设计模式-Flyweight享元模式(2)

推荐排行榜

1. C++设计模式-Observer观察者模式(5)
2. C++设计模式-Composite组合模式(4)
3. C++设计模式-Bridge桥接模式(3)
4. C++设计模式-Singleton(3)
5. C++设计模式-Factory工厂模式(2)



抽象基类及接口:

- 1、Abstraction::Operation(): 定义要实现的操作接口
- 2、AbstractionImplement::Operation(): 实现抽象类Abstraction所定义操作的接口, 由其具体派生类ConcreteImplementA、ConcreteImplementA或者其他派生类实现。
- 3、在Abstraction::Operation()中根据不同的指针多态调用AbstractionImplement::Operation()函数。

理解:

Bridge用于将表示和实现解耦,两者可以独立的变化.在Abstraction类中维护一个AbstractionImplement类指针,需要采用不同的实现方式的时候只需要传入不同的AbstractionImplement派生类就可以了.

Bridge的实现方式其实和Builde十分的相近,可以这么说:本质上是一样的,只是封装的东西不一样罢了.两者的实现都有如下的共同点:

抽象出来一个基类,这个基类里面定义了共有的一些行为,形成接口函数(对接口编程而不是对实现编程),这个接口函数在Buildier中是BuildePart函数在Bridge中是Operation函数;

其次,聚合一个基类的指针,如Builder模式中Director类聚合了一个Builder基类的指针,而Brige模式中Abstraction类聚合了一个AbstractionImplement基类的指针(优先采用聚合而不是继承);

而在使用的时候,都把对这个类的使用封装在一个函数中,在Bridge中是封装在Director::Construct函数中,因为装配不同部分的过程是一致的,而在Bridge模式中则是封装在Abstraction::Operation函数中,在这个函数中调用对应的AbstractionImplement::Operation函数.就两个模式而言,Builder封装了不同的生成组成部分的方式,而Bridge封装了不同的实现方式.

桥接模式就将实现与抽象分离开来,使得RefinedAbstraction依赖于抽象的实现,这样实现了依赖倒转原则,而不管左边的抽象如何变化,只要实现方法不变,右边的具体实现就不需要修改,而右边的具体实现方法发生变化,只要接口不变,左边的抽象也不需要修改。

常用的场景

- 1.当一个对象有多个变化因素的时候,考虑依赖于抽象的实现,而不是具体的实现。如上面例子中手机品牌有2种变化因素,一个是品牌,一个是功能。
- 2.当多个变化因素在多个对象间共享时,考虑将这部分变化的部分抽象出来再聚合/合成进来,如上面例子中的通讯录和游戏,其实是可以共享的。
- 3.当我们考虑一个对象的多个变化因素可以动态变化的时候,考虑使用桥接模式,如上面例子中的手机品牌是变化的,手机的功能也是变化的,所以将他们分离出来,独立的变化。

优点

- 1.将实现抽离出来,再实现抽象,使得对象的具体实现依赖于抽象,满足了依赖倒转原则。
- 2.将可以共享的变化部分,抽离出来,减少了代码的重复信息。
- 3.对象的具体实现可以更加灵活,可以满足多个因素变化的要求。

缺点

- 1.客户必须知道选择哪一种类型的实现。

设计中有超过一维的变化我们就可以用桥模式。如果只有一维在变化,那么我们用继承就可以圆满的解决问题。

代码如下:

Abstraction.h



```
1 #ifndef _ABSTRACTION_H_
2 #define _ABSTRACTION_H_
3
4 class AbstractionImplement;
5
6 class Abstraction
7 {
8 public:
9     virtual void Operation()=0; //定义接口, 表示该类所支持的操作
10    virtual ~Abstraction();
11 protected:
12    Abstraction();
13 };
14
15 class RefinedAbstractionA:public Abstraction
16 {
17 public:
18     RefinedAbstractionA(AbstractionImplement* imp); //构造函数
19     virtual void Operation(); //实现接口
20     virtual ~RefinedAbstractionA(); //析构函数
21 private:
22     AbstractionImplement* _imp; //私有成员
23 };
24
25 class RefinedAbstractionB:public Abstraction
26 {
27 public:
28     RefinedAbstractionB(AbstractionImplement* imp); //构造函数
29     virtual void Operation(); //实现接口
30     virtual ~RefinedAbstractionB(); //析构函数
31 private:
32     AbstractionImplement* _imp; //私有成员
```

```
33 };  
34 #endif
```



Abstraction.cpp




```
1 #include "Abstraction.h"  
2 #include "AbstractionImplement.h"  
3 #include <iostream>  
4  
5 using namespace std;  
6  
7 Abstraction::Abstraction()  
8 {}  
9  
10 Abstraction::~Abstraction()  
11 {}  
12  
13 RefinedAbstractionA::RefinedAbstractionA(AbstractionImplement* imp)  
14 {  
15     this->_imp = imp;  
16 }  
17  
18 RefinedAbstractionA::~RefinedAbstractionA()  
19 {  
20     delete this->_imp;  
21     this->_imp = NULL;  
22 }  
23  
24 void RefinedAbstractionA::Operation()  
25 {  
26     cout << "RefinedAbstractionA::Operation" << endl;
```

```
27     this->_imp->Operation();
28 }
29
30 RefinedAbstractionB::RefinedAbstractionB(AbstractionImplement* imp)
31 {
32     this->_imp = imp;
33 }
34
35 RefinedAbstractionB::~~RefinedAbstractionB()
36 {
37     delete this->_imp;
38     this->_imp = NULL;
39 }
40
41 void RefinedAbstractionB::Operation()
42 {
43     cout << "RefinedAbstractionB::Operation" << endl;
44     this->_imp->Operation();
45 }
```



AbstractImplement.h



```
1 #ifndef _ABSTRACTIONIMPLEMENT_H_
2 #define _ABSTRACTIONIMPLEMENT_H_
3
4 //抽象基类, 定义了实现的接口
5 class AbstractionImplement
6 {
7 public:
8     virtual void Operation()=0; //定义操作接口
9     virtual ~AbstractionImplement();
```

```
10 protected:
11     AbstractionImplement();
12 };
13
14 // 继承自AbstractionImplement,是AbstractionImplement的不同实现之一
15 class ConcreteAbstractionImplementA:public AbstractionImplement
16 {
17 public:
18     ConcreteAbstractionImplementA();
19     void Operation();//实现操作
20     ~ConcreteAbstractionImplementA();
21 protected:
22 };
23
24 // 继承自AbstractionImplement,是AbstractionImplement的不同实现之一
25 class ConcreteAbstractionImplementB:public AbstractionImplement
26 {
27 public:
28     ConcreteAbstractionImplementB();
29     void Operation();//实现操作
30     ~ConcreteAbstractionImplementB();
31 protected:
32 };
33 #endif
```



AbstractImplement.cpp



```
1 #include "AbstractionImplement.h"
2 #include <iostream>
3
4 using namespace std;
```



```
5
6 AbstractionImplement::AbstractionImplement()
7 {}
8
9 AbstractionImplement::~~AbstractionImplement()
10 {}
11
12 ConcreteAbstractionImplementA::ConcreteAbstractionImplementA()
13 {}
14
15 ConcreteAbstractionImplementA::~~ConcreteAbstractionImplementA()
16 {}
17
18 void ConcreteAbstractionImplementA::Operation()
19 {
20     cout << "ConcreteAbstractionImplementA Operation" << endl;
21 }
22
23 ConcreteAbstractionImplementB::ConcreteAbstractionImplementB()
24 {}
25
26 ConcreteAbstractionImplementB::~~ConcreteAbstractionImplementB()
27 {}
28
29 void ConcreteAbstractionImplementB::Operation()
30 {
31     cout << "ConcreteAbstractionImplementB Operation" << endl;
32 }
```



main.cpp



```
1 #include "Abstraction.h"
2 #include "AbstractionImplement.h"
3 #include <iostream>
4
5 using namespace std;
6
7 int main()
8 {
9     /* 将抽象部分与它的实现部分分离, 使得它们可以独立地变化
10
11     1、抽象Abstraction与实现AbstractionImplement分离;
12
13     2、抽象部分Abstraction可以变化, 如new RefinedAbstractionA(imp)、new RefinedAbstractionB(imp2);
14
15     3、实现部分AbstractionImplement也可以变化, 如new ConcreteAbstractionImplementA(), new
16     ConcreteAbstractionImplementB();
17
18     */
19     AbstractionImplement* imp = new ConcreteAbstractionImplementA();           //实现部分
20     ConcreteAbstractionImplementA
21     Abstraction* abs = new RefinedAbstractionA(imp);                         //抽象部分RefinedAbstractionA
22     abs->Operation();
23     cout << "-----" << endl;
24
25     AbstractionImplement* imp1 = new ConcreteAbstractionImplementB();         //实现部分
26     ConcreteAbstractionImplementB
27     Abstraction* abs1 = new RefinedAbstractionA(imp1);                       //抽象部分RefinedAbstractionA
28     abs1->Operation();
29     cout << "-----" << endl;
30
31     AbstractionImplement* imp2 = new ConcreteAbstractionImplementA();         //实现部分
```

```
ConcreteAbstractionImplementA
32     Abstraction* abs2 = new RefinedAbstractionB(imp2);           //抽象部分RefinedAbstractionB
33     abs2->Operation();
34
35     cout << "-----" << endl;
36
37     AbstractionImplement* imp3 = new ConcreteAbstractionImplementB(); //实现部分
ConcreteAbstractionImplementB
38     Abstraction* abs3 = new RefinedAbstractionB(imp3);           //抽象部分RefinedAbstractionB
39     abs3->Operation();
40
41     cout << endl;
42     return 0;
43 }
```



代码说明：

Bridge模式将抽象和实现分别独立实现，在代码中就是Abstraction类和AbstractionImplement类。

使用组合（委托）的方式将抽象和实现彻底地解耦，这样的好处是抽象和实现可以分别独立地变化，系统的耦合性也得到了很好的降低。

GoF的那句话中的“实现”该怎么去理解：“实现”特别是和“抽象”放在一起的时候我们“默认”的理解是“实现”就是“抽象”的具体子类的实现，但是这里GoF所谓的“实现”的含义不是指抽象基类的具体子类对抽象基类中虚函数（接口）的实现，是和继承结合在一起的。而这里的“实现”的含义指的是怎么去实现用户的需求，并且指的是通过组合（委托）的方式实现的，因此这里的实现不是指的继承基类、实现基类接口，而是指的是通过对象组合实现用户的需求。

实际上上面使用Bridge模式和使用带来问题方式的解决方案的根本区别在于是通过继承还是通过组合的方式去实现一个功能需求。

备注：

由于实现的方式有多种，桥接模式的核心就是把这些实现独立出来，让他们各自变化。

将抽象部分与它的实现部分分离：实现系统可能有多角度（维度）分类，每一种分类都可能变化，那么就把这种多角度分离出来让它们独立变化，减少它们之间的耦合。

在发现需要多角度去分类实现对象，而只用继承会造成大量的类增加，不能满足开放-封闭原则时，就要考虑用Bridge桥接模式了。

合成/聚合复用原则：尽量使用合成/聚合，精良不要使用类继承。

优先使用对象的合成/聚合将有助于保持每个类被封装，并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物。

分类: [面向对象](#), [设计模式](#)

标签: [设计模式](#), [Bridge桥接模式](#)



戒色

关注 - 0

粉丝 - 51

+加关注

« 上一篇: [C++设计模式-Prototype原型模式](#)

» 下一篇: [C++设计模式-Adapter适配器模式](#)

posted on 2013-07-01 17:40 [戒色](#) 阅读(9494) 评论(2) [编辑](#) [收藏](#)

3

0

Feedback

#1楼 2013-07-02 09:23 [钱吉](#)

楼主写的不错哦，推荐几本学习c++设计模式的书吧？只要经典的，编程语言为c++的，不要Java的

[支持\(0\)](#) [反对\(0\)](#)

#2楼 2013-12-27 21:43 [叶枫子](#)

@ DarkHorse

直接看四人帮的设计模式就行了。

[支持\(0\)](#) [反对\(0\)](#)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【推荐】如何快速搭建人工智能应用？

【活动】AI技术全面场景化落地实践

【大赛】2018首届“顶天立地”AI开发者大赛



最新IT新闻:

- 3年亏损超20亿 宝宝树IPO背后盈利待考
 - 易到又“怼”乐视：不正常关联交易，还冻结易到账户影响司机提款
 - 研发支出仅占营业收入8% 华大基因科研核心被夸大？
 - 股讯 | 美国科技股普跌 微软逆市涨近2%
 - 摩根大通：竞争对手增多 特斯拉股价在年底之前或暴跌超过40%
- » 更多新闻...



最新知识库文章:

- [观察之道：带你走进可观察性](#)
- [危害程序员职业生涯的三大观念](#)
- [断点单步跟踪是一种低效的调试方法](#)
- [测试 | 让每一粒尘埃有的放矢](#)
- [从Excel到微服务](#)
- » [更多知识库文章...](#)

Powered by:
[博客园](#)
Copyright © 戒色