动态多态

首先，动态多态是需要在运行期间，通过虚函数表才能确定具体的实现类型。不能够在编译期就确定

静态多态

CRTP(curious recurring template pattern)

```
#include <cstdio>


template <class Derived>
struct Base { void name() { (static_cast<Derived*>(this))->impl(); } };
struct D1 : public Base<D1> { void impl() { std::puts("D1::impl()"); } };
struct D2 : public Base<D2> { void impl() { std::puts("D2::impl()"); } };

void test()
{
    Base<D1> b1; b1.name();
    Base<D2> b2; b2.name();
    D1 d1; d1.name();
    D2 d2; d2.name();
}

int main()
{
    test();
}
```

output:

```
D1::impl()
D2::impl()
D1::impl()
D2::impl()
```

自己写一个例子：

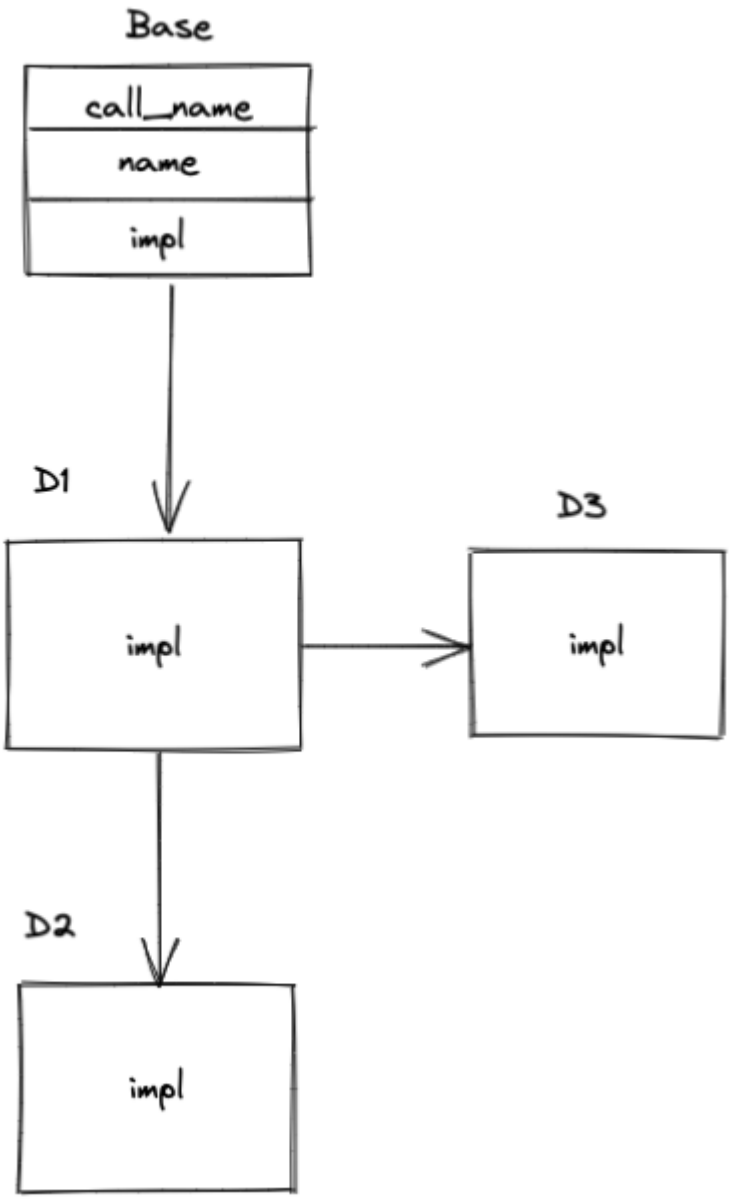
```
#include <cstdio>
template <class Derived>
struct Base {
    void call_name() {
        this->name();
    }
```

```
    void impl() {
      std::puts("Base::impl()");
    }
    void name() { (static_cast<Derived*>(this))->impl(); }
};

template <class Derived>
struct D1 : public Base<Derived> { void impl() { std::puts("D1::impl()"); } };

struct D2 : public D1<D2> { void impl() { std::puts("D2::impl()"); } };

struct D3 : public D1<D3> { };

void test()
{
    D2 d2;
    d2.call_name();
    D3 d3;
    d3.call_name();
}

int main()
{
    test();
}
```

output:

```
D2::impl()
D1::impl()
```

继承图如下：




```
static_cast<Derived*>(this))->impl();
```

本质是将Base的this转换模板参数Derved

实际Derived是由最上层（子类）传入自己，此时Impl函数会由最上层（子类）依次往下找到第一个覆盖的impl函数。

由于D3并没有自己覆盖Impl函数，所以D3执行的是它的父类D1::impl

更多例子

```
#include <cstdio>
```

```
template <class Derived>
struct Base {
    void call_name() {
        this->name();
    }
    void name() { (static_cast<Derived*>(this))->impl(); }

};

template <class Derived>
struct D1 : public Base<Derived> {
  void impl() {
    std::puts("D1::impl()");
  }
};

struct D2 : public D1<D2> {
  void impl() {
    std::puts("D2::impl()");
    this->D1::impl();
  }
};


void test(){
    D2 d2;
    d2.name();
}

int main(){
    test();
}
```

output:

```
D2::impl()
D1::impl()
```