☑ 写文章

### C/C++杂谈: CRTP benchmark

月踏 十 关注他 郭大海等 46 人赞同了该文章 前文《C/C++杂谈: CRTP》立了个flag要做实验测一下CRTP的实际性能,今天终于抽时间填了 之前挖的这个坑,本文用google benchmark做了一些实验来验证前文中的分析,下面来详细看

一、设计测试代码

收起

二、Debug模式的测试结果

目录

三、Release模式的测试结果 • 四、最后

下实验过程和结果。 实验类主要基于前文中用到的一个基类Base和两个子类Derived1、Derived2修改而来,每个子类 各有自己的foo、bar函数实现,我使用下面三种做法来做对比:

1. Normal模式:子类的foo、bar使用普通的函数实现 2. Virtual模式: foo、bar使用虚函数实现

3. CRTP模式: foo、bar使用CRTP方法实现 一、设计测试代码

#### Normal模式使用普通的函数实现,这样函数就没有了多态的能力,但是实现起来做简单,源代码 如下:

protected:

同子类对象的函数,源代码如下:

class NormalBase {

```
int64_t cnt_{0};
 };
 class NormalDerived1 : public NormalBase {
 public:
  int64_t foo(int64_t n) { cnt_ += n; return cnt_; }
  int64_t bar(int64_t n) { cnt_ += n; return cnt_; }
 class NormalDerived2 : public NormalBase {
 public:
  int64_t foo(int64_t n) { cnt_ += n + 0x0F; return cnt_; }
  int64_t bar(int64_t n) { cnt_ += n + 0xFF; return cnt_; }
 };
 static void test_normal(benchmark::State &state) {
  NormalDerived1 *ptr1 = new NormalDerived1;
  NormalDerived2 *ptr2 = new NormalDerived2;
  while (state.KeepRunning()) {
    for (auto i = 0; i < 32; i++) {
      auto v1 = ptr1 - sfoo(i);
      auto v2 = ptr2 - sfoo(v1);
      auto v3 = ptr1->bar(v2);
      auto v4 = ptr2->bar(v3);
  }
  delete ptr1;
   delete ptr2;
 BENCHMARK(test_normal);
Virtual模式引入了虚函数,这样就可以通过虚函数机制实现使用同一类型的基类指针动态调用不
```

class VirtualBase { public: virtual ~VirtualBase() = default;

```
virtual int64_t figo(int64_tsn) == 0; ilt
  virtual int64_t bara(int664_fto(in):64_t0; = 0;
                    virtual int64_t bar(int64_t n) = 0;
protected:
  int64_t cnt_{0}; int64_t cnt_{0};
class VirtualDerived1VirtualDerived1VirtualBase38
public:
                    virtual int64_t foo(int64_t n) override final { cnt_ += n; return cnt_; }
  virtual int64_t ficoalimt64_btar(m)64overoxidedeftimallorft_catt_returncht_return cnt_; }
 };
class VirtualDerived2alintpublic(inVirtualBasce final { cnt_ += n + 0x0F; return cnt_; }
                    virtual int64_t bar(int64_t n) override final { cnt_ += n + 0xFF; return cnt_; }
public:
  virtual int64_t; foo(int64_t,n) override final { cnt_ += n + 0x0F; return cnt_;
  virtual int64_timbar(int64_t(\n)unvermideinfoinal) {\recnt_ptr+=pan(n+)0xFF; return cnt_;
                   static void test_virtual(benchmark::State &state) {
};
int64_t virtual_foo(VirtualBase *ptr, int64 t n) { return ptr->foo(n); }
int64_t virtual_bar(VirtualBase_*ptn, int64_t n) { return ptr->bar(n); }
static void test_vinotual(benchmark:::State &state) {
  VirtualBase *ptr1 = new VirtualDerived1;
  VirtualBase *ptr2 = new VirtualDerived2;
 while (state KeepRunning (pt)2-{bar(v3);
    for (auto i = 0; i < 32; i++) {
      auto v1 = ptr1_7 foo (i); virtual foo(ptr2, v1)
      auto v2 = ptr2/\sqrt{300} (\sqrt{1})/\sqrt{100} (\sqrt{1})
      auto v3 = ptr14>bar(\sqrt{2})vjrtual_bar(ptr2, v3);
      auto v4 = ptr2->bar(v3);
                   delete ptr1;
      // auto v1 =dvirtual_foo(ptr1, i);
      // auto v2 = virtual foo(ptr2, v1);
      // auto v3 = virtual_bar(ptr1, v2);
      // auto v4 = virtual_bar(ptr2, v3);
  delete ptr1;
  delete ptr2;
BENCHMARK(test_virtual);
```

#### public: int64\_t foo(int64\_t n) { return static\_cast<T \*>(this)->internal\_foo(n); }

template<typename T>

class CRTPBase {

CRTP模式使用编译时多态的特性,实现了和动态多态相近的功能,源代码如下:

```
int64_t bar(int64_t n) { return static_cast<T *>(this)->internal_bar(n); }
 protected:
   int64 t cnt {0};
      public:
 class CRTPDerived14-inpublic CRTPBase<CRTPDerived1> (1);
              _t bar(int64_t n) {    return static_cast<T *>(this)->internal_bar(n);    }
 public:
   int64_ite4internal_foo(int64_t n) { cnt_ += n; return cnt_;}
   int64_t internal_bar(int64_t n) { cnt_ += n; return cnt_;}
          lass CRTPDerived1 : public CRTPBase<CRTPDerived1> {
 class CRTPDerited2foo(public) (CRTPBase CRTPDerived2> {
 public: int64_t internal_bar(int64_t n) { cnt_ += n; return cnt_;}
   int64't internal foo(int64 t n) { cnt += n + 0x0F; return cnt_;}
   int64_tticinternal_bar(int64_t n) {    cnt_ += n + 0xFF;    return cnt_;}
         int64_t internal_foo(int64_t n) { cnt_ += n + 0x0F; return cnt_;}
 template <typename T> Int64_t crtp_foo(CRTPBase<T> *ptr, int64_t n) { return ptr-
 template <typename T> int64 t.crtp_bar(CBTPBase<T>4*ptr,etint64 t.n) { return ptr-
 static two idtetesptenement (bienochmank bastates & state) tr, {nt64_t n) { return ptr->bar(n); }
   auto ptic1 in new cR PDe never tate & state) {
   auto ptr2 = new CRTPDerived2;
   while whatatelekeepRumming()) {
     for (3u^{2}b^{t_0}i)^{\frac{1}{2}}=0;0;3^{2};4+32;i++) {
        auto v1 = ptr1 - stoo(i);
        autout2v = ptr2+atoo(v1);
        autouto3√4= pt21≥50ap(v2);
        auto v4 = ptr2->bar(v3);
            // auto v2 = crtp_foo(ptr2, v1);
        // autotovB = \text{contop}_{\underline{I}} \text{fob}(\beta) \text{tr1, i);}
        // auto v2 = crtp_foo(ptr2, v1);
        // auto v3 = crtp_bar(ptr1, v2);
        //deautptrv4 = crtp_bar(ptr2, v3);
         delete ptr2;
        BENCHMARK(test_crtp);
   delete ptr1;
   deleterpth2ARK_MAIN();
 BENCHMARK(test_crtp);
 BENCHMARK_MAIN();
二、Debug模式的测试结果
使用Debug模式编译源码,使用的g++ flag如下:
```

## 测试结果如下:

Run on (8 X 2300 MHz CPU s)

L1 Instruction 32 KiB (x4)

L2 Unified 512 KiB (x4)

L1 Data 48 KiB (x4)

对%rbp和%rsp的操作开销:

movq %rdi, -8(%rbp)

movq %rsi, -16(%rbp)

测试结果如下:

L1 Instruction 32 KiB (x4)

L2 Unified 512 KiB (x4)

L3 Unified 8192 KiB (x1)

// foo、bar继续调用internal\_foo、internal\_bar

CPU Caches:

-00 -g -ggdb -Wall -std=c++14

```
L3 Unified 8192 KiB (x1)
Load Average: 1.91, 2.10, 2.23
          Time CPU Iterations
Benchmark
          202 ns
test_normal
                          202 ns
                                   3489949
test_virtual 236 ns
                                   3142410
                          234 ns
              350 ns
                                   2043635
test_crtp
                          349 ns
在编译器不做优化的情况下,测试结果有点出乎意料,CRTP的实现方法居然慢了这么多,反汇
编之后发现,原因是多了一次函数调用,即用户先调用了如下的foo、bar,然后foo、bar又继续
调用了internal_foo、internal_bar,函数调用的最基本开销在《HPC: X86-64 Assembly常用知
识点整理》中已经分析过,函数短小的话,主要是函数的Prologue和Epilogue的开销,也就是
```

void foo() { static\_cast<T \*>(this)->internal\_foo(); } void bar() { static\_cast<T \*>(this)->internal\_bar(); } // foo函数的汇编示例 pushq %rbp movq %rsp, %rbp subq \$16, %rsp

```
movq -8(%rbp), %rdi
movq -16(%rbp), %rsi
callq 0x100003eea <dyld_stub_binder+0x100003eea>
 addq $16, %rsp
popq %rbp
 retq
nopw %cs:(%rax,%rax)
nopl (%rax)
除了CRTP的benchmark和预想不一样之外,上面实验结果的Virtual模式比Normal模式稍慢,这
个是符合预期的,原因也很简单,虚函数的调用需要通过vtbl来间接进行。
三、Release模式的测试结果
使用Release模式编译源码,使用的g++ flag如下:
 -03 -Wall -std=c++14
```

Run on (8 X 2300 MHz CPU s) CPU Caches: L1 Data 48 KiB (x4)

```
Load Average: 5.20, 3.09, 2.72
                 Time
                              CPU Iterations
 Benchmark
               41.3 ns
 test_normal
                           41.2 ns
                                    13363114
                                     5558732
 test_virtual
               124 ns
                            124 ns
                                    16707760
 test_crtp
               40.0 ns
                           40.0 ns
从上面结果看出,CRTP模式和Normal模式差不多,Virtual模式最慢,这是符合预期的,使用和
Normal模式差不多的代价,就可以获得Virtual模式的多态效果,这时看CRTP模式的相关反汇编
代码的话,可以发现从foo到internal_foo的这层调用消失了,这其实也是inlining的结果,foo短
小精悍,是inlining的最佳候选,关于inlining,在之前的《HPC:表达式模板》中有过详细的分
```

四、最后 本文的实验和之前的分析是一致的,如果可以使用静态多态来替换动态多态的话,CRTP确实能 提升程序的性能,但说到底CRTP有可能省下的只是函数调用的开销,如果说函数体是计算密集 型的,这时候函数调用的开销几乎可以忽略不计,那么这时候CRTP就不一定是最好的选择,在

实际的场景中,具体使用哪种方案还是要具体情况具体分析。

# Benchmark

2条评论

编辑于 2021-11-13 15:12

C / C++

析,这里不再说了。

写评论 | 郭大海 关注了作者

默认 时间

```
函数体本身时间就很长的话,就直接虚函数最方便上层的架构设计。反之可以用模板,当然
  这也需要架构比较复杂需要保持灵活性的情况,如果是很简单的就直接普通写法。我理解的
  对吧
  2021-10-10
                                              ● 回复 🖿 3
  Tex
  可以测试下PGO,开启GCC的-fdevirtualize-speculatively来试试guarded devirtualization,
  跑下benchmark与CRTP比较下。这里理论上来说会比CRTP慢一点点,因为编译器会加条件
  语句从而带来开销。
                                              ● 回复 💧 2
  2021-10-19
```

## 文章被以下专栏收录

C/C++杂谈 C/C++杂谈