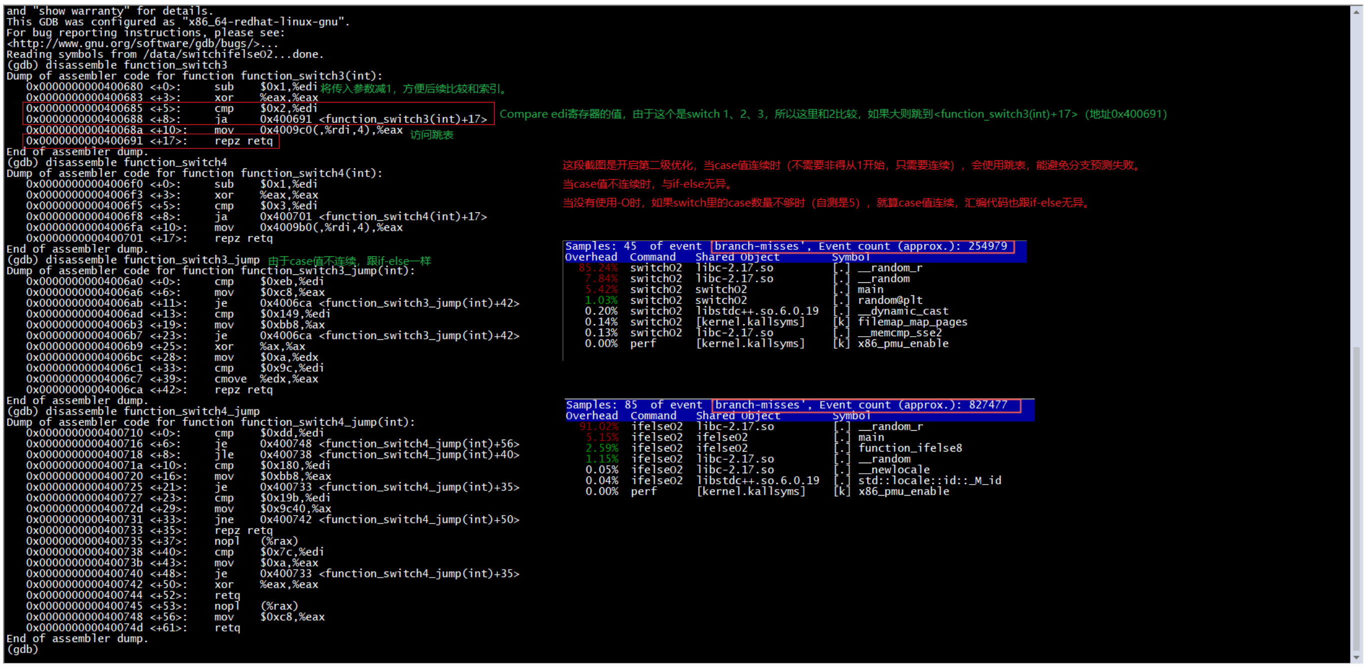
compare switch with ifelse



false sharing

大部分时候业务逻辑代码的优化是大头，但false sharing同样重要，当业务逻辑优化到一定程度后，这一块会是性能瓶颈。
就游戏服务器来说，大部分场景我觉得做成多进程单线程才是最优解，能避免使用锁或者false sharing造成的性能浪费。
如下截图是一个简单的测试代码，在64位机器上测试false sharing带来的消耗。
未做填充时的代码以及输出截图：

```cpp
#include <pthread.h>
#include <iostream>
#include <chrono>
using namespace std;

struct T
{
        int a;
        int b;
};

T global;

static void *thread_start1(void *arg)
{
        do
        {
                ++global.a;
        }while(global.a < 100000000);
        return nullptr;
}

static void* thread_start2(void* arg)
{
        do
        {
                ++global.b;
        }while(global.b < 100000000);
        return nullptr;
}

int main()
{
        global.a = 0;
        global.b = 0;
        auto start = std::chrono::steady_clock::now();
        pthread_t threadid1;
        pthread_t threadid2;
        pthread_create(&threadid1, nullptr, &thread_start1, nullptr);
        pthread_create(&threadid2, nullptr, &thread_start2, nullptr);

        pthread_join(threadid1, nullptr);
        pthread_join(threadid2, nullptr);
        auto end = std::chrono::steady_clock::now();
        auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
        std::cout << "Elapsed time: " << elapsed.count() << " ms\n";
        return 0;
}
```

```
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 2026 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 1886 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 2048 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 1755 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 2384 ms
```


做填充时的代码以及输出截图：

```
#include <pthread.h>
#include <iostream>
#include <chrono>
using namespace std;

struct T
{
        int a;
        int full[15];
        int b;
};

T global;

static void *thread_start1(void *arg)
{
        do
        {
                ++global.a;
        }while(global.a < 100000000);
        return nullptr;
}

static void* thread_start2(void* arg)
{
        do
        {
                ++global.b;
        }while(global.b < 100000000);
        return nullptr;
}

int main()
{
        global.a = 0;
        global.b = 0;
        auto start = std::chrono::steady_clock::now();
        pthread_t threadid1;
        pthread_t threadid2;
        pthread_create(&threadid1, nullptr, &thread_start1, nullptr);
        pthread_create(&threadid2, nullptr, &thread_start2, nullptr);

        pthread_join(threadid1, nullptr);
        pthread_join(threadid2, nullptr);
        auto end = std::chrono::steady_clock::now();
        auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
        std::cout << "Elapsed time: " << elapsed.count() << " ms\n";
        return 0;
}
```

```
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 90 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 129 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 134 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 98 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 93 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$
```


内存序

本想对C++11几种内存序做些记录，但实际准备写的时候又觉得没必要，首先是不管开发还是实际线上运营都是在X86_64架构下，而且一般都是加锁。其次这块感觉也没什么难点需要特别记录的。之所以明确表明X86_64架构，是因为X86_64是TSO（强一致性内存模型），这种CPU只有Store Buffer，没有Invalid Queue。也就是说这种内存模型只有store load这种情况会出现乱序，也就在这种情况下如果需要保证一致性才需要加内存屏障。
测试案例如下所示：
volatile关键字，表明变量是易变的，CPU每次都从主存上拿数据。
没有内存屏障代码截图以及测试结果如下图所示：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

volatile int x = 0, y = 0;
volatile int r1 = 0, r2 = 0;

void *thread1(void *arg) {
    x = 1;
    asm volatile("" ::: "memory"); //避免指令重排保证先执行x=1，再执行r1=y
    //__sync_synchronize();//内存屏障
    r1 = y;
    return NULL;
}

void *thread2(void *arg) {
    y = 1;
    asm volatile("" ::: "memory");//避免指令重排，保证先执行y=1，再执行r2=x
    //__sync_synchronize();//内存屏障
    r2 = x;
    return NULL;
}

int main() {
    int i, detected = 0;

    for (i = 0; i < 1000000; i++) {
        pthread_t t1, t2;

        x = 0; y = 0;
        r1 = 0; r2 = 0;

        pthread_create(&t1, NULL, thread1, NULL);
        pthread_create(&t2, NULL, thread2, NULL);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        if (r1 == 0 && r2 == 0) {
            detected++;
        }
    }

    printf("Detected store buffer behavior %d times.\n", detected);
    return 0;
}
```

```
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 2 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 7 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 5 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 1 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 1 times.
```

有内存屏障代码截图以及测试结果如下图所示：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

volatile int x = 0, y = 0;
volatile int r1 = 0, r2 = 0;

void *thread1(void *arg) {
    x = 1;
    //asm volatile("" ::: "memory"); //避免指令重排保证先执行x=1, 再执行r1=y
    __sync_synchronize();//内存屏障
    r1 = y;
    return NULL;
}

void *thread2(void *arg) {
    y = 1;
    //asm volatile("" ::: "memory");//避免指令重排，保证先执行y=1, 再执行r2=x
    __sync_synchronize();//内存屏障
    r2 = x;
    return NULL;
}

int main() {
    int i, detected = 0;

    for (i = 0; i < 1000000; i++) {
        pthread_t t1, t2;

        x = 0; y = 0;
        r1 = 0; r2 = 0;

        pthread_create(&t1, NULL, thread1, NULL);
        pthread_create(&t2, NULL, thread2, NULL);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        if (r1 == 0 && r2 == 0) {
            detected++;
        }
    }

    printf("Detected store buffer behavior %d times.\n", detected);
    return 0;
}
```

```
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
```

virtual table

```
class A
{
  public:
    virtual void func1(){}
};
class B
{
  public:
    virtual void func2(){}
};
class C : public A, public B
{
};
```

在64位linux平台上，C的大小毋庸置疑是16个字节。两个虚函数指针，但这两个虚函数指针指向的是哪呢？网上有不少讲这块的博客，基本上都说是指向两个虚函数表。那问题来了，是哪两个虚函数表？类A和类B的虚函数表？那类C的虚函数表呢？测试结果如下所示：

```
(gdb) p c
$1 = {<A> = {_vptr.A = 0x555555557cc0 <vtable for C+16>}, <B> = {_vptr.B = 0x555555557cd8 <vtable for C+40>}, <No data fields>}
(gdb) x /32xg 0x555555557cb0
0x555555557cb0 <_ZTV1C>:        0x0000000000000000      0x0000555555557d48
0x555555557cc0 <_ZTV1C+16>:     0x0000555555555268      0xfffffffffffffff8
0x555555557cd0 <_ZTV1C+32>:     0x0000555555557d48      0x0000555555555278
0x555555557ce0 <_ZTV1B>:        0x0000000000000000      0x0000555555557d80
0x555555557cf0 <_ZTV1B+16>:     0x0000555555555278      0x0000000000000000
0x555555557d00 <_ZTV1A+8>:      0x0000555555557d90      0x0000555555555268
0x555555557d10 <_ZTI1D>:        0x00007ffff7faad58      0x0000555555556005
0x555555557d20 <_ZTI1D+16>:     0x0000000200000000      0x0000555555557d90
0x555555557d30 <_ZTI1D+32>:     0x0000000000000002      0x0000555555557d80
0x555555557d40 <_ZTI1D+48>:     0x0000000000000802      0x00007ffff7faad58
0x555555557d50 <_ZTI1C+8>:      0x0000555555556008      0x0000000200000000
0x555555557d60 <_ZTI1C+24>:     0x0000555555557d90      0x0000000000000002
0x555555557d70 <_ZTI1C+40>:     0x0000555555557d80      0x0000000000000802
0x555555557d80 <_ZTI1B>:        0x00007ffff7faa008      0x000055555555600b
0x555555557d90 <_ZTI1A>:        0x00007ffff7faa008      0x000055555555600e
0x555555557da0: 0x0000000000000001      0x0000000000000001
```

先说结论，C的两个函数指针都是指向的vtable for C，只是指向的位置不一样。A的虚函数表起始地址是0x555555557cf0， B的虚函数表起始地址是0x555555557ce0， C的虚函数表起始地址是0x555555557cb0。可以看到C的虚函数表中具体存了哪些数据，首先在0x555555557cb8的值是0x0000555555557d48, 0x0000555555557d48也可以从表中得出结论是C的typeinfo信息。然后是0x0000555555555268，该地址的反汇

编代码如下所示：

```
(gdb) disassemble 0x0000555555555268
Dump of assembler code for function A::func1():
   0x0000555555555268 <+0>:     endbr64
   0x000055555555526c <+4>:     push   %rbp
   0x000055555555526d <+5>:     mov    %rsp,%rbp
   0x0000555555555270 <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000555555555274 <+12>:    nop
   0x0000555555555275 <+13>:    pop    %rbp
   0x0000555555555276 <+14>:    retq
End of assembler dump.
```

然后又是0x0000555555557d48，最后是0x0000555555555278，反汇编代码如下所示：

```
(gdb) disassemble 0x0000555555555278
Dump of assembler code for function B::func2():
   0x0000555555555278 <+0>:     endbr64
   0x000055555555527c <+4>:     push   %rbp
   0x000055555555527d <+5>:     mov    %rsp,%rbp
   0x0000555555555280 <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000555555555284 <+12>:    nop
   0x0000555555555285 <+13>:    pop    %rbp
   0x0000555555555286 <+14>:    retq
End of assembler dump.
```

由于C没有重写func1()和func2()，所以C里的函数地址和A、B的是一样的。综上所述，从逻辑上看是指向两个不同的虚函数表。从物理视角看是指向同一个虚函数表区域的不同部分。

make_shared

```cpp
template<typename _Tp, _Lock_policy _Lp = __default_lock_policy,
         typename... _Args>
  inline __shared_ptr<_Tp, _Lp>
  __make_shared(_Args&&... __args)
  {
    typedef typename std::remove_const<_Tp>::type _Tp_nc;
    return std::__allocate_shared<_Tp, _Lp>(std::allocator<_Tp_nc>(),
                                            std::forward<_Args>(__args)...);
  }
```

```cpp
template<typename _Tp, _Lock_policy _Lp = __default_lock_policy,
         typename _Alloc, typename... _Args>
  inline __shared_ptr<_Tp, _Lp>
  __allocate_shared(const _Alloc& __a, _Args&&... __args)
  {
    static_assert(!is_array<_Tp>::value, "make_shared<T[]> not supported");

    return __shared_ptr<_Tp, _Lp>(_Sp_alloc_shared_tag<_Alloc>{__a},
                                  std::forward<_Args>(__args)...);
  }
```

```cpp
// This constructor is non-standard, it is used by allocate_shared.
template<typename _Alloc, typename... _Args>
  __shared_ptr(_Sp_alloc_shared_tag<_Alloc> __tag, _Args&&... __args)
  : _M_ptr(), _M_refcount(_M_ptr, __tag, std::forward<_Args>(__args)...)
  { _M_enable_shared_from_this_with(_M_ptr); }
```

```cpp
template<typename _Tp, typename _Alloc, typename... _Args>
  __shared_count(_Tp*& __p, _Sp_alloc_shared_tag<_Alloc> __a,
                 _Args&&... __args)
  {
    using _Tp2 = __remove_cv_t<_Tp>;
    using _Sp_cp_type = _Sp_counted_ptr_inplace<_Tp2, _Alloc, _Lp>;
    typename _Sp_cp_type::__allocator_type __a2(__a._M_a);
    auto __guard = std::__allocate_guarded(__a2);
    _Sp_cp_type* __mem = __guard.get();
    auto __pi = ::new (__mem)
      _Sp_cp_type(__a._M_a, std::forward<_Args>(__args)...);
    __guard = nullptr;
    _M_pi = __pi;
    __p = __pi->_M_ptr();
  }
```