5.1 内存模型基础

这里从两方面来讲内存模型：一方面是基本结构，这与事务在内存中是怎样布局的有关；另一方面就是并发。对于并发基本结构很重要，特别是在低层原子操作。所以我将会从基本结构讲起。 c++ 中它与所有的对象和内存位置有关。

5.1.1 对象和内存位置

在一个 c++ 程序中的所有数据都是由对象(objects)构成。这不是说你可以创建一个int的衍生类，或者是基本类型中存在有成员函数，或是像在Smalltalk和Ruby语言下讨论程序那样——“一切都是对象”。“对象”仅仅是对C++数据构建块的一个声明。 c++ 标准定义类对象为“存储区域”，但对象还是可以将自己的特性赋予其他对象，比如，其类型和生命周期。

像int或float这样的对象就是简单基本类型；当然，也有用户定义类的实例。一些对象(比如，数组，衍生类的实例，特殊（具有非静态数据成员）类的实例)拥有子对象，但是其他对象就没有。

无论对象是怎么样的一个类型，一个对象都会存储在一个或多个内存位置上。每一个内存位置不是一个标量类型的对象，就是一个标量类型的子对象，比如，unsigned short、my_class*或序列中的相邻位域。当你使用位域，就需要注意：虽然相邻位域中是不同的对象，但仍视其为相同的内存位置。如图5.1所示，将一个struct分解为多个对象，并且展示了每个对象的内存位置。

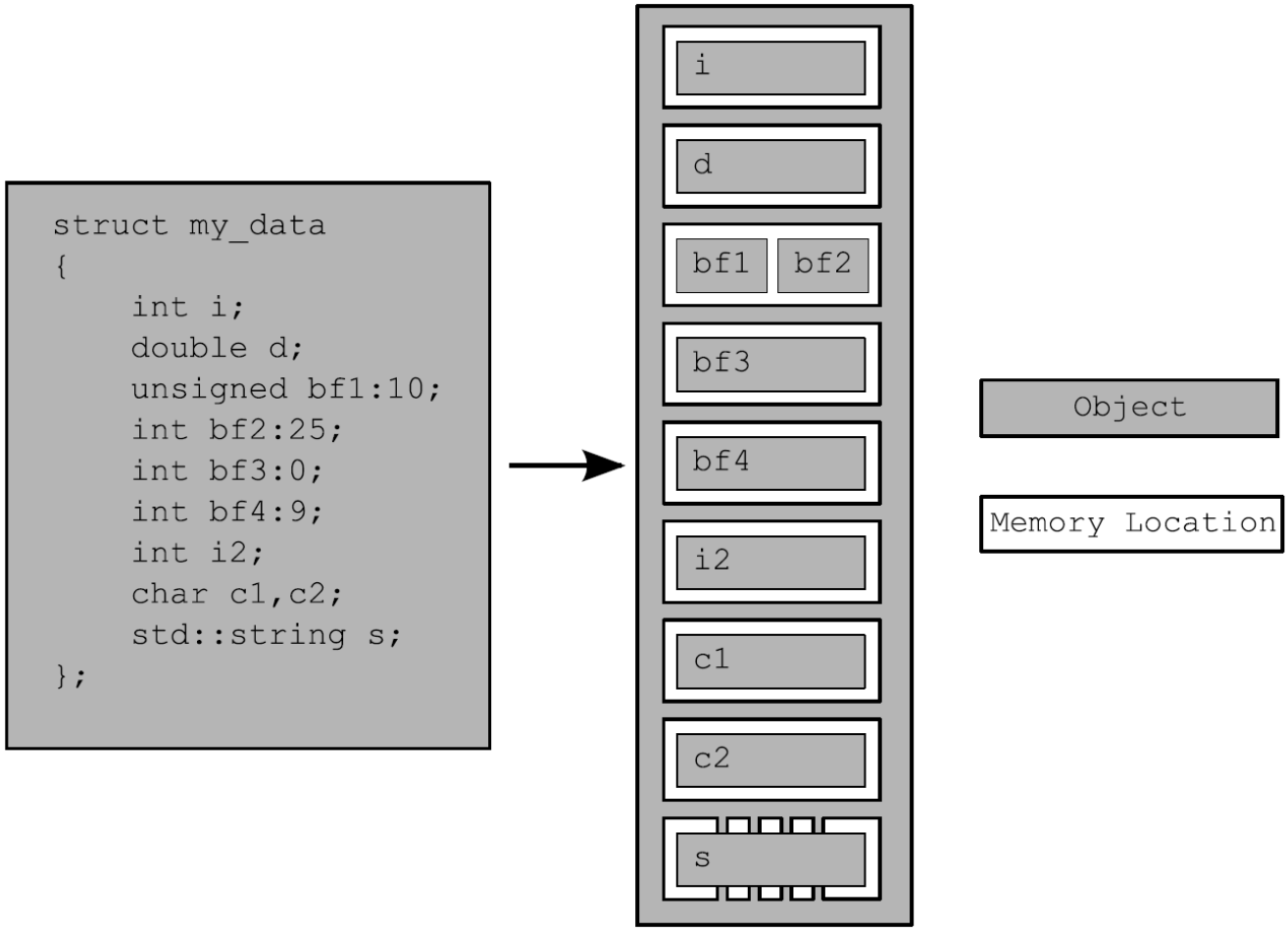

图5.1 分解一个struct，展示不同对象的内存位置

首先，完整的struct是一个有多个子对象(每一个成员变量)组成的对象。位域bf1和bf2共享同一个内存位置(int是4字节、32位类型)，并且 `std::string` 类型的对象s由内部多个内存位置组成，但是其他的每个成员都拥有自己的内存位置。注意，位域宽度为0的bf3是如何与bf4分离，并拥有各自的内存位置的。(译者注：图中bf3是一个错误展示，在 `C++` 和C中规定，宽度为0的一个未命名位域强制下一位域对齐到其下一type边界，其中type是该成员的类型。这里使用命名变量为0的位域，可能只是想展示其与bf4是如何分离的。有关位域的更多可以参考wiki的页面)。

这里有四个需要牢记的原则：

1. 每一个变量都是一个对象，包括作为其成员变量的对象。
2. 每个对象至少占有一个内存位置。
3. 基本类型都有确定的内存位置(无论类型大小如何，即使他们是相邻的，或是数组的一部分)。
4. 相邻位域是相同内存中的一部分。

我确定你会好奇，这些在并发中有什么作用，那么下面就让我们来见识一下。

5.1.2 对象、内存位置和并发

这部分对于 C++ 的多线程应用来说是至关重要的：所有东西都在内存中。当两个线程访问不同的内存位置时，不会存在任何问题，一切都工作顺利。而另一种情况下，当两个线程访问同一个内存位置，你就要小心了。如果没有线程更新内存位置上的数据，那还好；只读数据不需要保护或同步。当有线程对内存位置上的数据进行修改，那就有可能会产生条件竞争，就如第3章所述的那样。

为了避免条件竞争，两个线程就需要一定的执行顺序。第一种方式，如第3章所述那样，使用互斥量来确定访问的顺序；当同一互斥量在两个线程同时访问前被锁住，那么在同一时间内就只有一个线程能够访问到对应的内存位置，所以后一个访问必须在前一个访问之后。另一种方式是使用原子操作同步机制(详见5.2节中对于原子操作的定义)，决定两个线程的访问顺序。使用原子操作来规定顺序在5.3节中会有介绍。当多于两个线程访问同一个内存地址时，对每个访问这都需要定义一个顺序。

如果不去规定两个不同线程对同一内存地址访问的顺序，那么访问就不是原子的；并且，当两个线程都是“作者”时，就会产生数据竞争和未定义行为。

以下的声明由为重要：未定义的行为是 C++ 中最黑暗的角落。根据语言的标准，一旦应用中有任何未定义的行为，就很难预料会发生什么事情；因为，未定义行为是难以预料的。我就知道一个未定义行为的特定实例，让某人的显示器起火的案例。虽然，这种事情应该不会发生在你身上，但是数据竞争绝对是一个严重的错误，并且需要不惜一切代价避免它。

另一个重点是：当程序中的对同一内存地址中的数据访问存在竞争，你可以使用原子操作来避免未定义行为。当然，这不会影响竞争的产生——原子操作并没有指定访问顺序——但原子操作把程序拉回了定义行为的区域内。

在我们了解原子操作前，还有一个有关对象和内存地址的概念需要重点了解：修改顺序。

5.1.3 修改顺序

每一个在C++程序中的对象，都有(由程序中的所有线程对象)确定好的修改顺序，在的初始化开始阶段确定。在大多数情况下，这个顺序不同于执行中的顺序，但是在给定的执行程序中，所有线程都需要遵守这顺序。如果对象不是一个原子类型(将在5.2节详述)，你必要确保有足够的同步操

作，来确定每个线程都遵守了变量的修改顺序。当不同线程在不同序列中访问同一个值时，你可能就会遇到数据竞争或未定义行为(详见5.1.2节)。如果你使用原子操作，编译器就有责任去替你做必要的同步。

这一要求意味着：投机执行是不允许的，因为当线程按修改顺序访问一个特殊的输入，之后的读操作，必须由线程返回较新的值，并且之后的写操作必须发生在修改顺序之后。同样的，在同一线程上允许读取对象的操作，要不返回一个已写入的值，要不在对象的修改顺序后(也就是在读取后)再写入另一个值。虽然，所有线程都需要遵守程序中每个独立对象的修改顺序，但它们没有必要遵守在独立对象上的相对操作顺序。在5.3.3节中会有更多关于不同线程间操作顺序的内容。

所以，什么是原子操作？它如何来规定顺序？接下来的一节中，会为你揭晓答案。