数据局部性

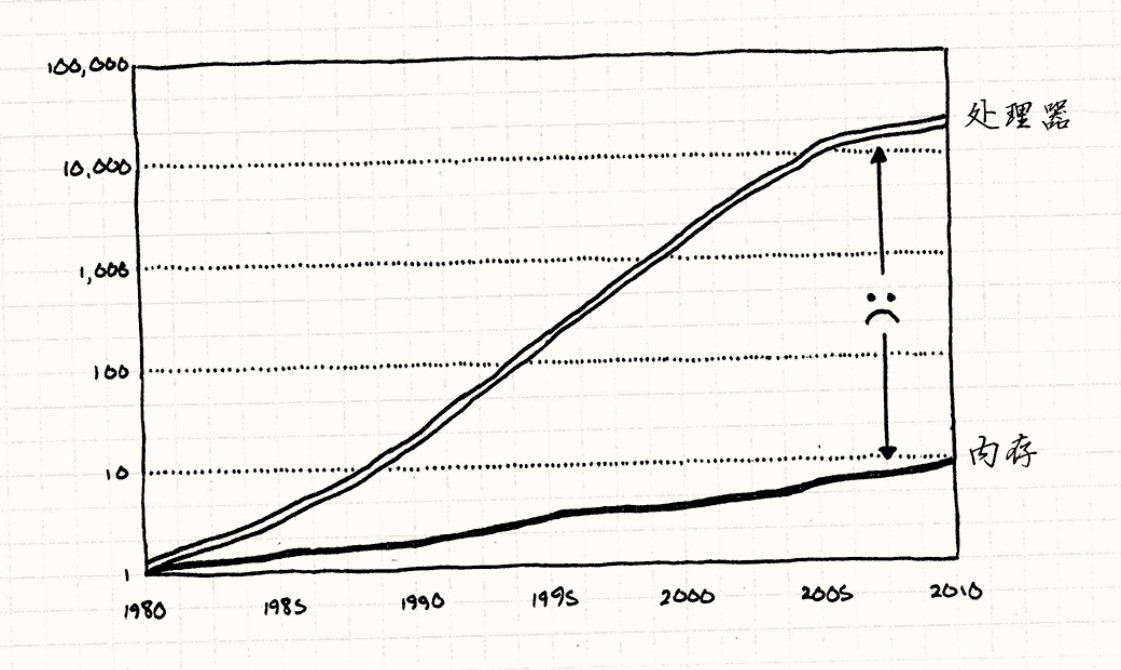
意图

合理组织数据，充分使用**CPU**的缓存来加速内存读取。

动机

我们被欺骗了。 他们一直向我们展示CPU速度每年递增的图表，就好像摩尔定律不是观察历史的结果，而是某种定理。 无需吹灰之力，软件凭借着新硬件就可以奇迹般地加速。

芯片确实越来越快（就算现在增长的速度放缓了），但硬件巨头没有提到某些事情。 是的，我们可以更快地处理数据，但不能更快地获得数据。



处理器和RAM的发展速度从1980开始不同。如你所见，CPU飞跃式发展，RAM读取速度被远远甩到了后面。

这个数据来自**Computer Architecture: A Quantitative Approach** 由John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau基于Tony Albrecht的“Pitfalls of Object-Oriented Programming”写就。

为了让你超高速的CPU刮起指令风暴， 它需要从内存获取数据加载到寄存器。 如你所知，RAM没有紧跟CPU的速度增长，差远了。

借助现代的硬件，要消耗上百个周期才能从RAM获得一比特的数据。 如果大部分指令需要的数据都需要上百个周期去获取， 那么为什么我们的CPU没有在99%的时间空转着等待数据？

事实上，等待内存读取确实会消耗很长时间，但是没有那么糟糕。 为了解释为什么，让我们看一看这一长串类比……

> 它被称为“乱序存储器（RAM，random access memory）”是因为， 不像光驱，理论上你从某块获取数据的速度和从其他块获取数据的速度是一样的。你不需要像光盘那样考虑连续读取。
>
> 或者，你**现在不需要**。就像接下来看到的，RAM不是那么乱序地读取。

数据仓库

想象一下，你是小办公室里的会计。 你的任务是拿盒文件，然后做一些会计工作——把数据加起来什么的。 你必须根据一堆只有会计能懂的晦涩逻辑，取出特定标记的文件盒并工作。

> 我也许不应该在例子中用我一无所知的职业打比方。

由于辛勤地工作，天生的悟性，还有进取心，你可以在一分钟内处理一个文件盒。 但是这里有一个小小的问题。所有这些文件盒都存储在分离的仓库中。 想要拿到一个文件盒，需要让仓库管理员带给你。 他开着叉车在传送带周围移动，直到找到你要的文件盒。

严格地说，这会消耗他一整天才能完成。 不像你，他下个月就不会被雇佣了。 这就意味着无论你有多快，一天只能拿到一个文件盒。 剩下的时间，你只能坐在那里，质疑自己怎么选了这个折磨灵魂的工作。

一天，一组工业设计师出现了。 他们的任务是提高操作的效率——比如让传送带跑得更快。 在看着你工作几天后，他们发现了几件事情：

- 通常，当你处理文件盒时，下一个需要处理的文件盒就在仓库同一个架子上。
- 叉车一次只取一个文件盒太愚蠢了。
- 在你的办公室角落里还是有些空余空间的。

> 访问刚刚访问的事物旁边的位置，描述这种行为的术语是**引用局部性。**

他们想出来一个巧妙的办法。 无论何时你问仓库要一个盒子，他都会带给你一托盘的盒子。 他给你想要的盒子，以及它周围的盒子。 他不知道你是不是想要这些（而且，根据工作条件，他根本不在乎）； 他只是尽可能地塞满托盘，然后带给你。

无视工作场地的安全，他直接将叉车开到你的办公室，然后将托盘放在办公室的角落。

当你需要新盒子，你需要做的第一件事就是看看它在不在办公室角落的托盘里。 如果在，很好！你只需要几分钟拿起它然后继续计算数据。 如果一个托盘中有五十个盒子，而幸运的你需要所有盒子，你可以以五十倍的速度工作。

但是如果你需要的盒子不在托盘上，就需要新的一托盘的盒子。 由于你的办公室里只能放一托盘的盒子，你的仓库朋友只能将旧的拿走，带给你一托盘全新的盒子。

CPU的托盘

奇怪的是，这就是现代CPU运转的方式。如果还不够明显，你是CPU。 你的桌子是CPU的寄存器，一盒文件就是寄存器能放下的数据。 仓库是机器的RAM，那个烦人的仓库管理员

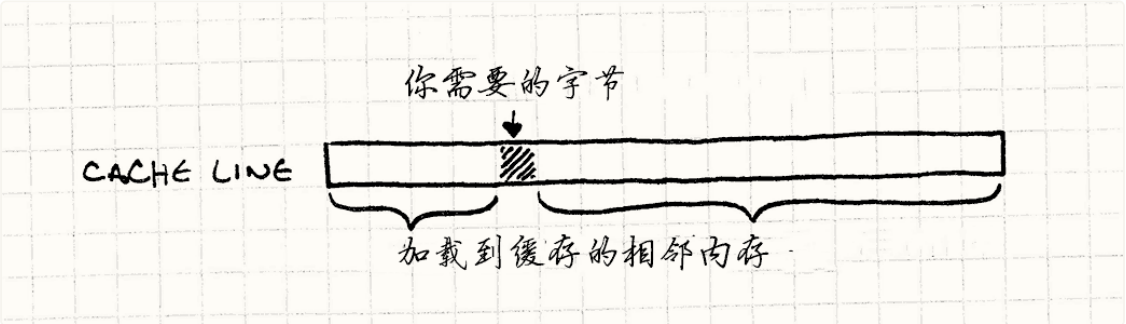
是从主存加载数据到寄存器的总线。

如果我在三十年前写这一章，这个比喻就到此为止了。 但是芯片越来越快，而RAM，好吧，“没有跟上”，硬件工程师开始寻找解决方案。 他们想到的是CPU缓存技术。

现代的电脑在芯片内部有一小块存储器。 CPU从那上面取数据比从内存取数据快得多。 它很小，因为需要放在芯片上，它很快，因为使用的（静态RAM，或称SRAM）内存更贵。

> 现代硬件有多层缓存，就是你听到的“L1”，“L2”，“L3”之类的。 每一层都更大也更慢。在这章里，我们不关心内存是不是多层的，但了解一下还是很有必要的。

这一小片内存被称为缓存（特别地，芯片上的被称为L1级缓存）， 在比喻中，它是由托盘扮演的。 无论何时芯片需要从RAM取一字节的数据，它自动将一整块内存读入然后将其放入缓存——通常是64到128字节。 这些一次性传输的字节被称为cache line。



如果你需要的下一字节数据就在这块上， CPU从缓存中直接读取，比从RAM中读取快得多。 成功从缓存中找到数据被称为“缓存命中”。 如果不能从中获得而得去主存里取，这就是一次缓存不命中。

> 我在类比中掩盖了（至少）一个细节。在办公室里，只能放一个托盘，或者一个cache line。 真实的缓存包含多个cache line。关于这点的细节超出了本书的范围，搜索“缓存关联性”来了解相关内容。

当缓存不命中时，CPU空转——它不能执行下一条指令，因为它没有数据。 它坐在那里，无聊地等待几百个周期直到取到数据。 我们的任务是避免这一点。想象你在优化一块性能攸关的游戏代码，长得像这样：

```
for (int i = 0; i < NUM_THINGS; i++)
{
  sleepFor500Cycles();
  things[i].doStuff();
}
```

你会做的第一个改动是什么？对了。 将那个无用的，代价巨大的函数调用拿出来。 这个调用等价于一次缓存不命中的代价。 每次跳到内存，都会延误你的代码。

等等，数据是性能？

当着手写这一章时，我花费了一些时间制作一个类似游戏的小程序，用于收集缓存使用的最好情况和最坏情况。 我想要缓存速度的基准，这样可以得到缓存失效造成的性能浪费情况。

当看到一些工作的结果时，我惊到了。 我知道这是一个大问题，但眼见为实。 两个程序完成完全相同的计算，唯一的区别是它们会造成缓存不命中的数量。 较慢的程序比较快的程序慢五十倍。

> 这里有很多警告。特别地，不同的计算机有不同的缓存设置，所以我的机器可能和你的不同， 专用的游戏主机与个人电脑不同，而二者都与移动设备不同。
>
> 你得自己测测看。

这让我大开眼界。我一直从代码的角度考虑性能，而不是数据。 一个字节没有快慢，它是静态的。但是因为缓存的存在，组织数据的方式直接影响了性能。

现在真正的挑战是将这些打包成一章内可以讲完的东西。 优化缓存使用是一个很大的话题。 我还没有谈到指令缓存呢。 记住，代码也在内存上，而且在执行前需要加载到CPU上。 有些更熟悉这个主题的人可以就这个问题写一整本书。

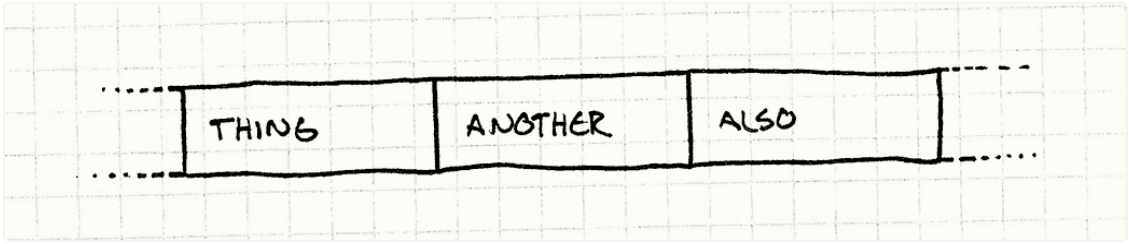
> 事实上，有人**确实**写了一本书：**Data-Oriented Design**，作者Richard Fabian.

既然你已经在阅读这本书了， 我有几个基本技术来帮你考虑数据结构是如何影响性能的。

这可以归结成很简单的事情：芯片读内存时总是获得一整块cache line。 你能从cache line读到越多你要的东西，速度就越快。 所以目标是组织数据结构，让要处理的数据紧紧相邻。

> 这里有一个核心假设：单线程。 如果在多线程上处理邻近数据，让它在多个**不同**的cache line上更好。 如果两个线程争夺同一cache line上的数据，两个核都得花些时间同步缓存。

换言之，如果你正处理`Thing`，然后`Another`然后`Also`，你需要它们这样呆在内存里：



注意，不是`Thing`，`Another`，和`Also`的指针，而是一个接一个的真实数据，。 CPU读到`Thing`，也会读取`Another`和`Also`（取决于数据的大小和cache line的大小）。 当你开始下一个时，它们已经在缓存上了。芯片很高兴，你也很高兴。

模式

现代的CPU有缓存来加速内存读取。 它可以更快地读取最近访问过的内存的毗邻内存。 通过提高内存局部性来提高性能——保证数据以处理顺序排列在连续内存上。

何时使用

就像大多数优化方案，使用数据局部性的第一准则是在遇到性能问题时使用。 不要将其应用在代码库不经常使用的角落上。 优化代码不会让你过得更轻松，因为其结果往往更加复杂，更加缺乏灵活性。

就本模式而言，还得确认你的性能问题确实由缓存不命中引发。 如果代码是因为其他原因而缓慢，这个模式帮不上忙。

简单的估算方案是手动添加指令，检查代码中两点间消耗的时间，寄希望于精确的计时器。为了找到糟糕的缓存使用，你需要使用更加复杂的东西。 你想要知道有多少缓存不命中，又是在哪里发生的。

幸运的是，有现成的工具做这些。 在数据结构上做大手术前，花一些时间了解这些工具是如何工作， 理解它们抛出的一大堆数据（令人惊讶地复杂）是很有意义的。

> 不幸的是，这些工具大部分不便宜。如果在一个主机开发团队，你可能已经有了使用它们的证书。
>
> 如果没有，一个极好的替代选项是Cachegrind。 它在模拟的CPU和缓存结构上运行你的程序，然后报告所有的缓存交互。

话虽这么说，缓存不命中仍会影响游戏的性能。 虽然不应该花费大量时间提前优化缓存的使用，但是在设计过程中仍要思考数据结构是不是对缓存友好。

记住

软件体系结构的特点之一是抽象。 这本书的很多章节都在谈论如何解耦代码块，这样可以更容易地进行改变。 在面向对象的语言中，这几乎总是意味着接口。

在C++中，使用接口意味着通过指针或者引用访问对象。 但是使用指针就意味在内存中跳跃，这就带来了这章想要避免的缓存不命中。

> 接口的另一半是**虚方法调用**。 这需要CPU查找对象的虚函数表，找到调用方法的真实指针。 所以，你又一次追踪指针，造成缓存不命中。

为了讨好这个模式，你需要牺牲一些宝贵的抽象。 你越围绕数据局部性设计程序，就越是在放弃继承、接口和它们带来的好处。 没有银弹，只有挑战性的权衡。这就是乐趣所在！

示例代码

如果你真的要一探数据局部性优化的深处，那么你会发现有无数的方法去分割数据结构，将其切为CPU更好处理的小块。 为了热热身，我会先从一些最通用的分割方法开始。 我们会在游戏引擎的特定部分介绍它们， 但是（像其他章节一样）记住这些通用方法也能在其他部分使用。

连续数组

让我们从处理一系列游戏实体的游戏循环开始。 实体使用了组件模式，被分解到不同的领域——AI，物理，渲染。 这里是GmaeEntity类。

```
class GameEntity
{
public:
  GameEntity(AIComponent* ai,
             PhysicsComponent* physics,
             RenderComponent* render)
  : ai_(ai), physics_(physics), render_(render)
  {}

  AIComponent* ai() { return ai_; }
  PhysicsComponent* physics() { return physics_; }
  RenderComponent* render() { return render_; }

private:
  AIComponent* ai_;
  PhysicsComponent* physics_;
  RenderComponent* render_;
};
```

每个组件都有相对较少的状态，也许只有几个向量或一个矩阵， 然后会有方法去更新它。这里的细节无关紧要，但是想象一下，大概是这样的：

> 就像名字暗示的，这些是更新方法[7]模式的例子。 甚至render()也是这个模式，只是换了个名字。

```
class AIComponent
{
public:
  void update() { /* 处理并修改状态…… */ }

private:
  // 目标，情绪，等等……
};

class PhysicsComponent
{
public:
  void update() { /* 处理并修改状态…… */ }

private:
  // 刚体，速度，质量，等等……
};

class RenderComponent
{
public:
  void render() { /* 处理并修改状态…… */ }

private:
  // 网格，纹理，着色器，等等……
};
```

游戏循环管理游戏世界中一大堆实体的指针数组。每个游戏循环，我们都要做如下事情：

1. 为每个实体更新他们的AI组件。
2. 为每个实体更新他们的物理组件。
3. 为每个实体更新他们的渲染组件。

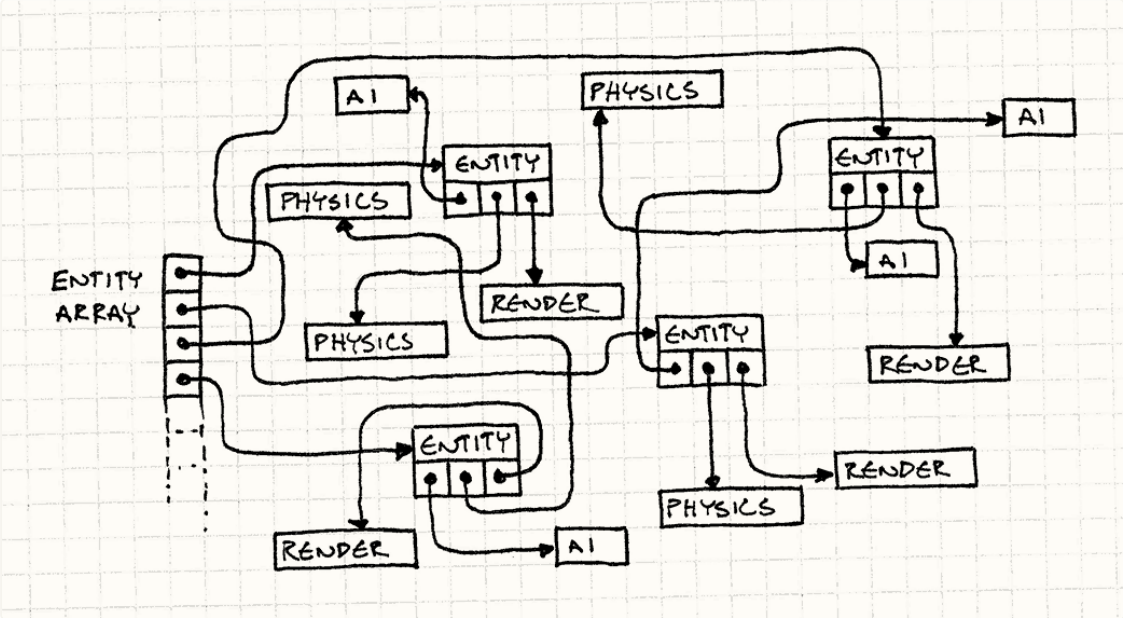
很多游戏引擎以这种方式实现：

```
while (!gameOver)
{
  // 处理AI
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->ai()->update();
  }

  // 更新物理
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->physics()->update();
  }

  // 绘制屏幕
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->render()->render();
  }

  // 其他和时间有关的游戏循环机制……
}
```

在你听说CPU缓存之前，这些看上去完全无害。 但是现在，你得看到这里有隐藏着的不对之处。 这不是在颠簸缓存，这是在四处乱晃然后猛烈地敲击。看看它做了什么：

1. 游戏实体的数组存储的是指针，所以为了获取游戏实体，我们得转换指针。缓存不命中。
2. 然后游戏实体有组件的指针。又一次缓存不命中。
3. 然后我们更新组件。
4. 再然后我们退回第一步，为游戏中的每个实体做这件事。

令人害怕的是，我们不知道这些对象是如何在内存中布局的。 我们完全任由内存管理器摆布。 随着实体的分配和释放，堆的组织会更乱。



> 每一帧，游戏循环得追踪这些指针来获取数据。

如果我们的目标是在游戏地址空间中四处乱转，完成“256MB内存四晚廉价游”，这也许是一个很好的决定。 但是我们的目标是让游戏跑得尽可能快，而在主存四处乱逛不是一个好办法。 记得`sleepFor500Cycles()`函数吗？上面的代码效率和它也差不多了。

> 描述浪费时间转换指针这一行为的术语是“追逐指针”，它并不像听上去那么有趣。

我们能做得更好。 第一个发现是，之所以跟着指针去寻找游戏实体，是因为可以立刻跟着另一个指针去获得组件。 `GameEntity`本身没有有意义的状态和有用的方法。组件 才是游戏循环需要的。

众多实体和组件不能像星星一样散落在黑暗的天空中，我们得脚踏实地。 我们将每种组件存入巨大的数组：一个数组给AI组件，一个给物理，另一个给渲染。

就像这样：

```
AIComponent* aiComponents =
    new AIComponent[MAX_ENTITIES];
PhysicsComponent* physicsComponents =
    new PhysicsComponent[MAX_ENTITIES];
RenderComponent* renderComponents =
    new RenderComponent[MAX_ENTITIES];
```

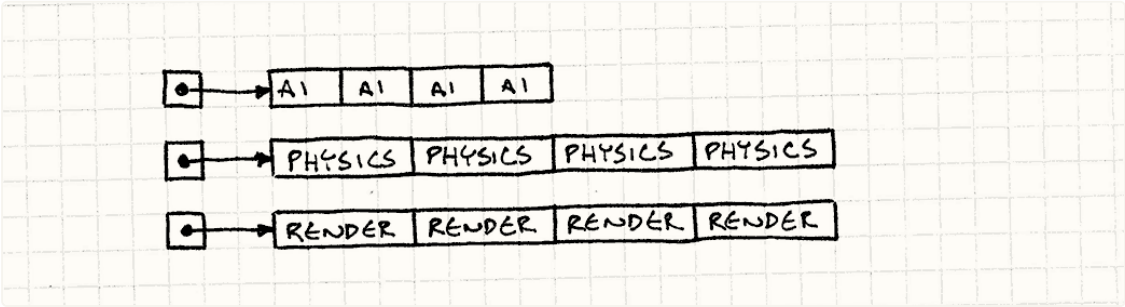
> 使用组件时，我最不喜欢的就是组件这个单词的长度。

让我强调一点，这些都是组件的数组，而不是指向组件的指针。数据都在那里一个接着一个排列。 游戏循环现在可以直接遍历它们了。

```
while (!gameOver)
{
  // 处理AI
  for (int i = 0; i < numEntities; i++)
  {
    aiComponents[i].update();
  }

  // 更新物理
  for (int i = 0; i < numEntities; i++)
  {
    physicsComponents[i].update();
  }

  // 绘制屏幕
  for (int i = 0; i < numEntities; i++)
  {
    renderComponents[i].render();
  }

  // 其他和时间有关的游戏循环机制……
}
```

> 在这里做得更好的一个技巧是新代码中有更少的->操作符。 如果你想要提高数据局部性，找找那些你可以摆脱的间接跳转。

我们消除了所有的指针追逐。不在内存中跳来跳去，而是直接在三个数组中做直线遍历。



这将一股字节流直接泵到了CPU饥饿的肚子里。 在我的测试中，这个改写后的更新循环是之前性能的**50**倍。

有趣的是，我们并没有在这里放弃太多的封装。 是的，游戏循环直接更新游戏组件而不是通过游戏实体，但在此之前它已经确保了以正确的顺序运行。 即使如此，每个组件的内部还是具有很好的封装性。 它们的封装性取决于自身的数据和方法。我们只是改变了使用它们的方法。

这也不意味着我们摆脱了`GameEntity`。它拥有它组件的指针这一状态仍然得以保持。 它的组件指针现在只是指到了这个数组之中。 对游戏的其他部分，如果你还是想传递一个“游戏实体”，一切照旧。 重要的是性能攸关的游戏循环部分回避了这点，直接获取数据。

打包数据

假设我们在做粒子系统。 根据上节的建议，将所有的粒子放在巨大的连续数组中。让我们用管理类封装它。

> `ParticleSystem`类是对象池的一个例子，通常为单一类型对象而构建。

```
class Particle
{
public:
  void update() { /* 重力，等等…… */ }
```

```
  // 位置，速度，等等……
};

class ParticleSystem
{
public:
  ParticleSystem()
  : numParticles_(0)
  {}

  void update();
private:
  static const int MAX_PARTICLES = 100000;

  int numParticles_;
  Particle particles_[MAX_PARTICLES];
};
```

系统中的基本更新方法看起来是这样的：

```
void ParticleSystem::update()
{
  for (int i = 0; i < numParticles_; i++)
  {
    particles_[i].update();
  }
}
```

但实际上不需要同时更新所有的粒子。 粒子系统维护固定大小的对象池，但是粒子通常不是同时在屏幕上活跃。 最简单的解决方案是这样的：

```
for (int i = 0; i < numParticles_; i++)
{
  if (particles_[i].isActive())
  {
    particles_[i].update();
  }
}
```

我们给`Particle`一个标志位来追踪其是否在使用状态。 在更新循环时，我们检查每个粒子的这个标志位。 这会将粒子其他部分的数据也加载到缓存中。 如果粒子没有在使用，那么跳过它去检查下一个。 这时粒子加载到内存中的其他数据都是浪费。

活跃的粒子越少，要在内存中跳过的部分就越多。 越这样做，在两次活跃粒子有效更新之间发生的缓存不命中就越多。 如果数组很大又有很多不活跃的粒子，我们又在颠簸缓存了。

如果连续数组中的对象不是连续处理的，实际上这个办法也没有太多效果。 如果有太多不活跃的对象需要跳过，就又回到了问题的起点。

> 理解底层代码的程序员也可以看出这里的问题。 使用`if`为每个粒子检查会引起**分支预测错误**和**流水线暂停**。 在现代CPU中，一条简单的“指令”实际消耗多个时钟周期。 为了保持CPU繁忙，指令**流水线化**，在前面的指令处理完成之前就开始处理后续指令。
>
> 为了实现流水线，CPU需要猜测接下来要执行哪一条指令。 在顺序结构的代码中，这很简单；但是加入控制流，就难了。 当它为`if`执行指令，它是猜粒子是活跃的然后执行`update()`调用，还是猜它不活跃呢？

> 为了回答这一点，芯片做**分支预测**——它看看之前的代码选择了哪条分支然后照做。 但是当循环不断在活跃的和不活跃的粒子之间转换，预测就失败了。
>
> 当它失败，CPU取消它推测的代码（**流水线更新**），从头开始。 这在机器上波及广泛，这就是为什么有时候你看到开发者在热点代码避免控制流。

鉴于本节的标题，你大概可以猜出答案是什么了。 我们不监测活跃与否的标签，我们根据标签排序粒子。 将所有活跃的粒子放在列表的前头。 如果知道了这些粒子都是活跃的，就不必再检查这些标识位了。

还可以很容易地追踪有多少活跃的粒子。这样，更新循环变成了这种美丽的东西：

```
for (int i = 0; i < numActive_; i++)
{
  particles[i].update();
}
```

现在没有跳过任何数据。 加载入缓存的每一字节都是需要处理的粒子的一部分。

当然，我可没说每帧都要对整个数组做快排。 这将抵消这里的收益。我们想要的是保持数组的顺序。

假设数组已经排好序了——开始时确实如此，因为所有粒子都不活跃——它变成未排序的时候即是粒子被激活或者被反激活时。 我们可以很轻易地处理这两种情况。 当一个粒子激活时，我们让它占据第一个不活跃粒子的位置， 将不活跃粒子移动到激活序列的尾端，完成一次交换：

```
void ParticleSystem::activateParticle(int index)
{
  // 不应该已被激活！
  assert(index >= numActive_);

  // 将它和第一个未激活的粒子交换
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;

  // 现在多了一个激活粒子
  numActive_++;
}
```

为了反激活粒子，只需做相反的事情：

```
void ParticleSystem::deactivateParticle(int index)
{
  // 不应该已被激活！
  assert(index < numActive_);

  // 现在少了一个激活粒子
  numActive_--;

  // 将它和最后一个激活粒子交换
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;
}
```

很多程序员（包括我在内）已经对于在内存中移动数据过敏了。 将一堆数据移来移去的消耗感觉比发送指针要大得多。 但是如果你加上了解析指针的代价，有时候这种估算是错误的。 在有些情况下，如果能够保证缓存命中，在内存中移动数据消耗更小。

在你做这种决策前要记得**验证**这点。

将粒子根据激活状态保持排序——就不需要给每个粒子都添加激活标志位了。 这可以由它在数组中的位置和`numActive_`计数器推断而得。 这让粒子对象更小，意味着在cache lines中能够打包更多数据，能跑得更快。

但是并非万事如意。 你可以从API看出，我们放弃了一定的面向对象思想。 `Particle`类不再控制其激活状态了。 你不能在它上面调用`activate()`因为它不知道自己的索引。 相反，任何想要激活粒子的代码都需要接触到粒子系统。

在这个例子中，将`ParticleSystem`和`Particle`这样牢牢绑一起没有问题。 我将它们视为两个物理类的的组合概念。 这意味着粒子只在特定的粒子系统中有意义。 在这种情况下，很可能是粒子系统在复制和销毁粒子。

冷/热 分割

这里是最后一种取悦缓存的技术例子。 假设某些游戏实体有AI控件。 其中包括一些状态——现在正在播放的动画，正在前往的方向，能量等级，等等——这些东西每帧都会发生变化。 就像这样：

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;
};
```

但它也有一些罕见事件的状态。 它存储了一些数据，描述它遭到猎枪痛击后会掉落什么战利品。 掉落数据在实体的整个生命周期只会使用一次，就在它结束的前一霎那：

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  // 之前的字段……
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

假设我们遵循前面的章节，那么当我们更新AI组件时，就穿过了一序列打包好的连续数组。 那个数据包含所有掉落物的信息。 这让每个组件都变得更大了，这就减少了我们能够加载到cache line中的组件个数。 每帧的每个组件都会将战利品数据加载到内存中去，即使我们根本不会去使用它。

这里的解决方案被称为“冷/热分割”。这个点子源于将数据结构划分为两个分离的部分。 第一部分保存“热”数据，那些每帧都要调用的数据。 剩下的片段被称为“冷”数据，在那里存储使用的次数较少的数据。

这里的热部分是AI组件的主体。 它是使用最多的部分，所以我们不希望解析指针去找到它。 冷组件可以被归到一边去，但是我们还是需要访问它，因此我们在热组件中包含一个

指向它的指针，就像这样：

```
class AIComponent
{
public:
  // 方法……
private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;

  LootDrop* loot_;
};

class LootDrop
{
  friend class AIComponent;
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

现在我们每帧都要遍历AI组件，加载到内存的数据只包含必需的数据（以及那个指向冷数据的指针）。

> 我们可以继续去除指针，为冷热数据维护平行数组。 仍能够为组件找到冷数据，因为两者在各自数组中索引值是相同的。

你可以看到事情是怎么变得模棱两可的。 在我的例子中，哪些是冷数据，哪些是热数据是很明确的，但是在真实的游戏中一般很少可以这么明显地分割。 如果你有一部分数据，实体在一种状态下会经常使用，另一种状态则不会，那该怎么办？ 如果实体只在特定关卡时使用一块特定的数据，又该怎么办？

做这种优化有时就是在走钢丝。 很容易陷入其中，消耗无尽的时间把数据挪来挪去看看性能如何。 需要通过实践来掌握在哪里付出努力。

设计决策

这章更接近于介绍一种思维定势——将数据的组织模式作为游戏性能的关键部分。 实际上具体的设计空间是开放的。 你可以让数据局部性影响整个架构，或者只在局部几个核心数据结构上使用这个模式。

最需要关心的问题是在何时何地使用这个模式，但是这里还有其他几个问题需要回答。

> Noel Llopis的著名文章让很多人围绕缓存设计游戏，他称之为“面向数据的设计”。

你如何处理多态？

到了现在，我们避开了子类和虚方法。 我们假设有打包好的同类对象。 这种情况下，我们知道它们有同样的大小。 但是多态和动态调用也是有用的工具。我们如何调和呢？

- 别这么干

  最简单的解决方案是避免子类，至少在做内存优化的部分避免使用。 无论如何，软件工程师文化已经和大量使用继承渐行渐远了。

  > 一种保持多态的灵活性而不使用子类的方法是借助于类型对象模式。

- 简洁安全。 你知道在处理什么类，所有的对象都是同样大小。
- 更快 动态调用意味着在跳转表中寻找方法，然后跟着指针寻找特定的代码。 这种消耗在不同硬件上区别很大，但动态调用总会带来一些代价。

> 就像往常一样，万事无绝对。 在大多数情况下，虚方法调用中C++编译器需要一次重定向。 但是在**某些**情况下，如果可以知道接受者的具体类型，编译器可以**去虚拟化**，然后静态地调用正确的方法。 去虚拟化在一些just-in-time虚拟机比如Java和JavaScript中更为常见。

- 不灵活 当然，使用动态调用的原因就是它提供了在不同对象间展示不同的行为的强大能力。 如果游戏想要不同的实体使用独特的渲染、移动或攻击，虚方法是处理它的好办法。 把它换成包含巨大的`switch`的非虚方法会超级慢。

- 为每种类型使用分离的数组：

  我们使用多态，这样即使不知道对象的类型，也能引入行为。 换言之，有了一包混合的东西，我们想要其中每个对象在接到通知时去做自己的事情。

  但是这提出来一个问题：为什么开始的时候要把它们混在一起呢？ 取而代之，为什么不为每种类型保持一个单独的集合呢？

  - 对象被紧密地排列着。 每个数组只包含同类的对象，这里没有填充或者其他的古怪。
  - 静态调度。 一旦获得了对象的类型，你不必在所有时候使用多态。你可以使用常规的非虚方法调用。
  - 得追踪每个集合。 如果你有很多不同类型，为每种类型分别管理数组可是件苦差事。
  - 得明了每一种类型。 由于你为每种类型管理分离的集合，你无法解耦类型集合。多态的魔力之一在于它是开放的——与一个接口交互的代码可以与实现此接口的众多类型解耦。

- 使用指针的集合：

  如果你不太担心缓存，这是自然的解法。 只要一个指针数组指向基类或者接口类型，你就获得了想要的多态，对象可以想多大就多大。

  - 灵活。这样构建集合的代码可以与任何支持接口的类工作。完全开放。
  - 对缓存不友好。 当然，我们在这里讨论其他方案的原因就是指针跳转导致的缓存不友好。 但是，记住，如果代码不是性能攸关的，这很有可能是行得通的。

游戏实体是如何定义的？

如果与组件模式串联使用此模式， 你会获得多个数组，包含组成游戏实体的组件。 游戏循环会在那里直接遍历它们，所以实体本身就不是那么重要了， 但是在其他你想要与“实体”交互的代码库部分，一个概念上的实体还是很有用的。

这里的问题是它该如何被表示？如何追踪这些组件？

- 如果游戏实体是拥有它组件指针的类：

  这是第一个例子中的情况。纯OOP解决方案。 你得到了`GameEntity`类，以及指向它拥有的组件的指针。 由于它们只是指针，我们并不知道这些组件是如何在内存中组织的。

- 你可以将实体存储到连续数组中。 既然游戏实体不在乎组件在哪里，你可以将组件好好打包，组织在数组中来优化遍历。
- 拿到一个实体，可以轻易地获得它的组件。 就在一次指针跳转后的位置。
- 在内存中移动组件很难。 当组件启用或者关闭时，你可能想要在数组中移动它们，保证启用的组件位于前列。 如果在实体中有指针指向组件时直接移动该组件，一不小心指针就会损毁。你得保证同时更新指向组件的指针。

- 如果游戏实体是拥有组件**ID**的类：

  使用裸指针的挑战在于在内存中移动它很难。你可以使用更加直接的方案：使用ID或者索引来查找组件。

  ID的实际查找过程是由你决定的，它可能很简单，只需为每个实体保存独特的ID，然后遍历数组查找， 或者更加复杂地使用哈希表，将ID映射到组件现有的位置。

  - 更复杂。 ID系统不是高科技，但是还是需要比指针多做些事情。你得实现它然后排除漏洞，这里需要消耗内存。
  - 更慢。 很难比直接使用指针更快。需要使用搜索或者哈希来帮助实体找到它的组件。
  - 你需要访问组件“管理器”。 基本思路是用抽象的ID标识组件。你可以使用它来获得对应组件对象的引用。 但是为了做到这点，你需要让ID有办法找到对应的组件。这正是包裹着整个连续组件数组的类所要做的。

    通过裸指针，如果你有游戏实体，你可以直接找到组件，而这种方式你需要接触游戏实体和组件注册器。

    > 你也许在想，“我会把它做成单例！问题解决！”好吧，在某种程度上是这样的。 不过，你也许想要先看看这章。

- 如果游戏实体本身就是一个**ID**：

  这是某些游戏引擎使用的新方式。一旦实体的行为和状态被移出放入组件，还剩什么呢？ 事实上，没什么了。实体干的唯一事情就是将组件连接在一起。 它的存在只是为了说明：这个AI组件和这个物理组件还有这个 渲染组件合起来， 定义了一个存在于游戏世界中的实体。

  这很重要，因为组件要相互交互。 渲染组件需要知道实体位于何处，而位置信息也许是物理组件的属性。 AI组件想要移动实体，因此它需要对物理组件施加力。每个组件都需要以某种方式获得同一实体中的其他组件。

  有些聪明人意识到你唯一需要的东西就是ID。不是实体知道组件，而是组件知道实体。每个组件都知道拥有它的实体的ID。当AI组件需要它所属实体的物理组件时，它只需要找到那个拥有同样ID的物理组件。

  你的实体类整个消失了，取而代之的是围绕数字的华丽包装。

  - 实体很小。当你想要传递游戏实体的引用时，只需一个简单的值。
  - 实体是空的。当然，将所有东西移出实体的代价是，你必须将所有东西移出。 不能再拥有组件独有的状态和行为，这样更加依赖于组件模式。
  - 不必管理实体的生命周期。 由于实体只是内置值类型，不需要被显式分配和释放。当它所有的组件都被释放时，对象就隐式“死亡”了。

- 查找实体的某一组件也许会很慢。 这和前一方案有相同的问题，但是是在另一个方向上。 为了找某个实体的组件，你需要给ID做对象映射。这一过程消耗也许很大。

  但是，这一次是性能攸关的。 在更新时，组件经常与它的兄弟组件交互，因此你需要经常地查找组件。 解法是让组件在数组中的索引作为实体的“ID”。

  如果每个实体都是拥有相同组件的集合，那么组件数组就是完全同步的。 组件数组三号位的AI组件与在物理组件数组三号位的组件相关联。

  但是，记住，这强迫你保持这些数组平行。 如果你想要用不同的方式排序或者打包它们就会变得很难。 你也许需要一些没有物理组件或者没有渲染组件的实体。 而它们仍保证与其他组件同步，没有办法独自排序物理组件数组和渲染组件数组。

参见

- 这一章大部分围绕着组件模式。 这种模式的数据结构绝对是为缓存优化的最常见例子。事实上，使用组件模式让这种优化变得容易了。 由于实体是按“领域”（AI，物理，等等）更新的，将它们划出去变成组件，更容易将它们保存为对缓存友好的合适大小。

  但是这不意味你只能为组件使用这个模式！ 任何需要接触很多数据的关键代码，考虑数据局部性都是很重要的。

- Tony Albrecht的 《Pitfalls of Object-Oriented Programming》 PDF 也许是最广为人知的内存友好游戏设计指南。它让很多人（包括我！）明白了数据结构对性能而言是多么重要。

- 几乎同时，Noel Llopis关于同一话题写了一篇 非常有影响力的博客。

- 这一模式几乎完全得益于同类对象的连续存储数组。 随着时间的推移，你也许需要向那个数组增加或删除对象。 对象池模式正是关于这一点。

- 游戏引擎Artemis是首个也是最著名的为游戏实体使用简单ID的游戏框架。