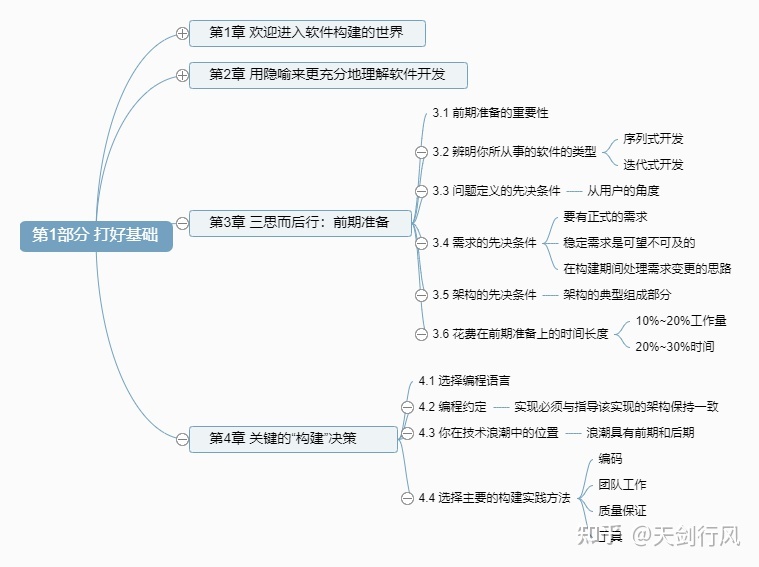
https://max.book118.com/html/2017/0506/104721656.shtm





做任何事情都需要前期准备，在软件开发中更是如此，尽管如此，还是有很多程序员接到任务后就是想着尽快编码，很多老板不重视软件开发的前期准备。要想保证一个软件的质量，在前期准备，需求分析，架构设计，编码，测试，维护等每一个环节都要重视质量。具体程序员接到任务的时候要检查一下在你之前的那些软件活动有没有准备好，如果需求中有好多没有说明的地方，架构设计也不明确，你不知道需要和其它模块之间如何通信，基础组件啥也没有，这种情况下进行详细设计和编码会很受罪。

前期准备所花费的时间是不容易把握的，也没有个固定的衡量标准，但前期准备是必须要做的，前期准备的根本目的是降低风险，提高项目质量

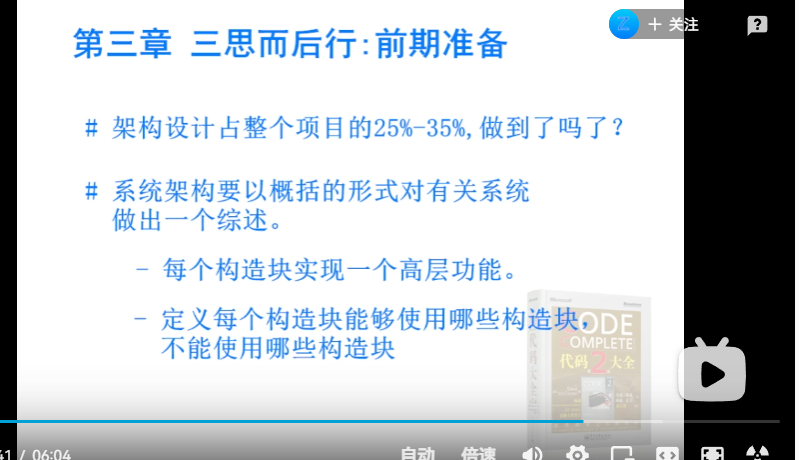
3.03

第三章讲的是前期准备

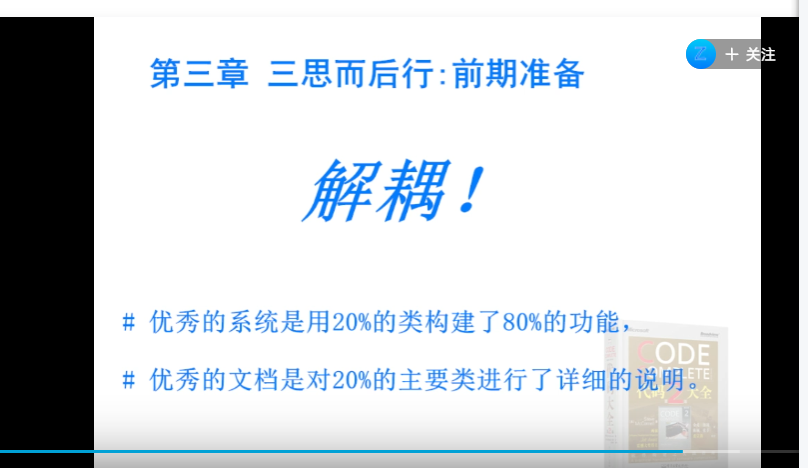
以前的理解是编代码之前的工作都叫前期准备，而本书的前期准备指的是架构的设计，按照书中的说法，这个架构的设计要占到项目的25%到35%的时间，其实我们可以反思一下，在以前的项目中，我们以前的项目的设计架构花费了多少时间，有没有达到25%，我觉得大部分是没有达到的，其实我们也遇到过，如果这个占比太低的话，我们设计出来的东西可能是边设计边开发，这样经常导致东一榔头西一棒子，没有任何的整体性可言，也谈不上什么叫优秀吧，所以我们需要对这个系统架构有一个认识。

在书中，系统架构的定义是要以概括的形式对有关系统做出一个综述。

具体来说，就是应该定义主要结构块，什么是结构块呢？其实就是高层的功能，抽象的功能，比如说web界面，数据库的访问等等这些模块。根据程序规模的不同，各个构造块可能是由一个类或者许多类组成的。架构还需要定义每个构造块可以使用，可以访问哪些构造块。不能使用不能访问哪些构造块。举个例子，我的web页面就不能直接访问数据模块，要想访问，就必须通过数据访问模块进行查询的操作。这样做非常好的一点就是清晰明了的构建了构造块的访问规则，

并且它最重要的一点是有助于程序的解耦，让程序的耦连不是非常大，便于后面的修改。接下来就是每个构造块下面可能有一个或者多个的类，我们同时也要定义好这些类的职责，就像构造块一样，以及他们之间的访问的规则。文章中有一句话说的很好，优秀的系统是用20%的类构建了80%的功能，

优秀的文档是对20%的主要类进行了详细的说明。我们经常会抱怨这个文档很难写。其实从这里看，我们的根本原因不是文笔不好也不是我们没有编写代码编写文档的习惯，而在于我们在系统架构上就出了问题，如果我们能够在架构的时候，就能够做到用20%的类就把大部分功能完成了，我们写文档就只用写这20%的类，这样我们的文档就不会写的很多，这样我们就有动力去写了，不会像现在总是在逃避，所以写文档最终的原因，一定出在架构的设计上。



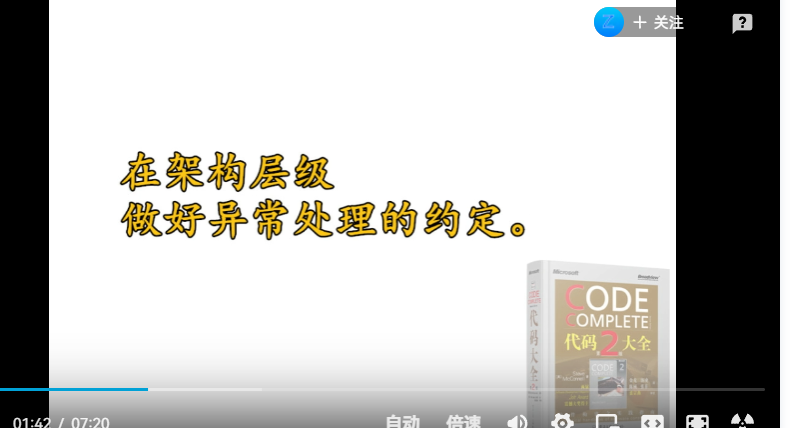
Ok来录制一下第三张下半段的读书笔记。下我在刚入行的时候经常被别人说代码的学院风特别重，其实我当时非常疑惑是到底什么是学院风，后来经过读书学习，我慢慢知道，其实学院风指的是一大类的代码风格，这类代码通常只完成了最基本的功能，只要把它放到复杂的环境中，就会暴露出各种各样的问题，比如程序崩溃，内存溢出等等。



但是这个环境它到底复杂在哪里了，导致我的程序经常崩溃。举个例子，在理想的环境下，程序的输入都是标准的，没有错误的，而在现实环境中程序的输入总是千奇百怪的，他不会按照我们的想法来输入，我们在处理这些不标准的输入时候，稍有不慎就会出现错误，所以如果能够做好错误处理，程序的稳定性便会大幅度的提高。

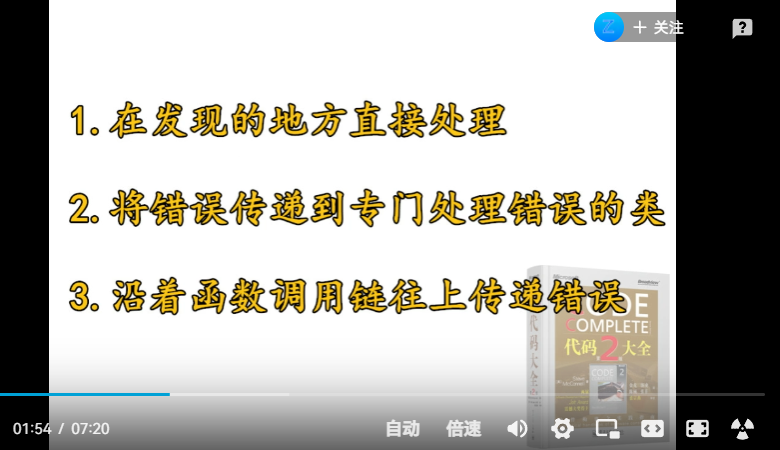


错误处理在现代的软件开发中是一个非常棘手的问题。当我们阅读优秀的商业级源码的时候啊，我们经常能看到，其实这些源码中90%的代码都是处理异常的，这意味着什么？它其实意味着一个学院风很重的程序员在自以为完成工作的时候，其实他只完成了10%核心的代码，他并没有写那90%处理异常的代码，我们可以想象一下，这样的代码把它放到实际环境中是多么的脆弱，



对于错误处理，我们最好在架构层次上进行约定，什么叫架构层次？

我来举个例子说明一下，比如说我们应当在哪里进行错误处理，



这里我们往往是有三种的方法，

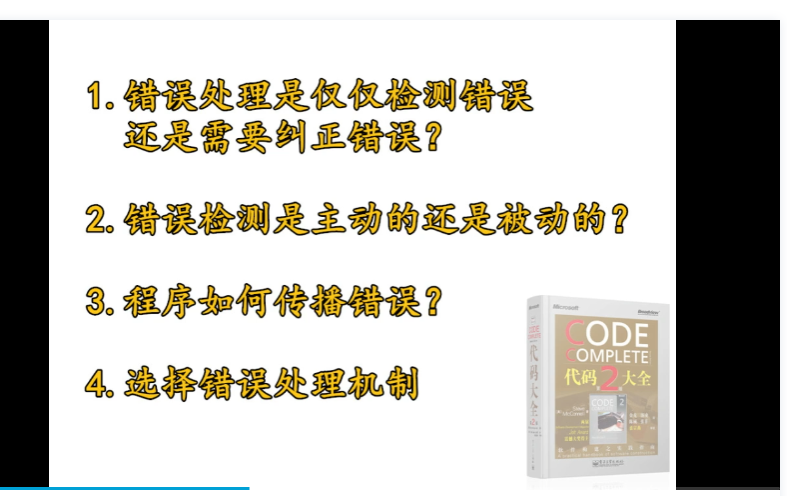
第一种是我在哪里发现我就在哪里处理，这是最常见的。

第二种，我们将单独设立一个错误处理的类，我们一旦遇到错误，我们就调用这个类进行错误处理。

还有一种方法是我们把这个错误我们不在本地处理，我们把它往上，在这个函数调用链上往上传递，让上层函数进行处理。



这样如果我们的开发小组没有进行事先约定好，那么在开发的过程中我们就必须进行临时的决定，而这些临时的决定往往是考虑不周到的，整个系统就是因为这些很多考虑不周到的决定，降低了它的健壮性，



除了约定错误处理的层次之外，还有以下4个约定是非常重要的。

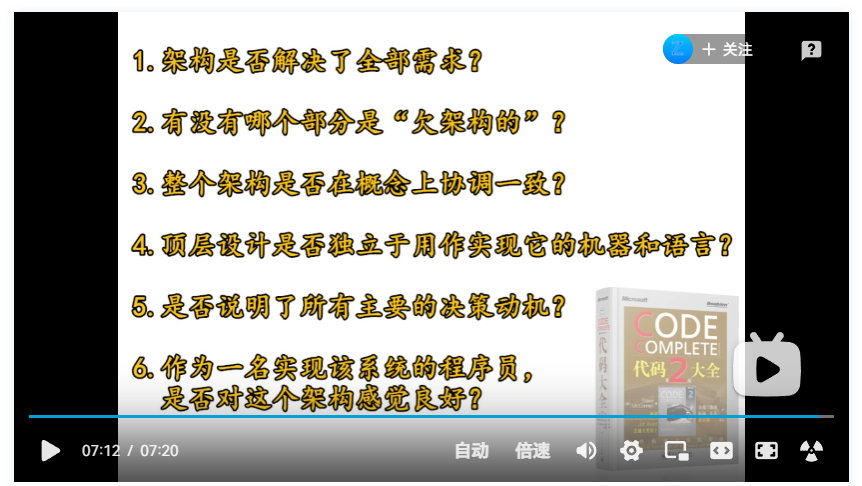
1. 低点错误处理是仅仅检测错误还是需要纠正错误？如果我们是仅仅检测错误，那么这个程序就必须像没有发生错误一样能够继续运行。如果是纠正错误，这个程序就必须能够从错误中恢复过来是吧？我们可以两种选一种。

第二个错误检测是主动的还是被动的，系统可以主动的预测错误，比如说我们在处理某个输入的时候，我们可以先检测一下他是否越界了，如果越界了，我们就“ber”报了一个错误，他是越界了，我们不处理了，我们也可以在发生错误的时候进行响应。比如我计算完一个结果，以后我发现这个结果是明显错误的，我们就抛弃这个结果，我们就不用这个结果了。

第三点程序是如何传播错误的。这里边主要有三种方法比较常用，第一种方法是直接丢弃异常的数据，这个就比较暴力了，它不一定适合每一种情况。第二种方法比较通用，是我们写一个错误处理的分支，我们遇到错误之后我就进入到分支中，我把错误处理好了，我再往下走。

第三种是我暂时先不管这个错误，等到我把所有的数据都处理完毕了，我在向用户通知我发生了一个错误，这三种方法都是可行的，我们都可以任选其一。

第四个约定，我们是建立一套自己的错误处理机制，还是运用系统自带的错误处理机制？这里边其实很多人他都推荐我们要自己建立一套错误处理机制，但是这里我是 不是很支持这样的想法的，为什么？因为我觉得windows已经很牛逼了，它已经存储了几万条的异常，你确定你能比他做的更完善吗？我觉得我不可能做的比windows更好，我的能力是有限的，所以我还是乖乖的用windows自己自带的这种错误处理机制。



Ok讲完错误处理，第三章就只剩下最后一个要点了，如何评价一个架构的总体质量？书中给出了6个评价点，我们一个来讲一讲，

第一点架构是否解决了全部需求，这个就很简单了，没有解决全部需求的架构肯定是一个不合格的架构。第二点，有没有哪个部分是欠架构的，是否明确宣布了在这方面的预期指标？什么是欠架构的？简单来说没有设计完善的那一部分就叫欠架构的部分。

在软件设计中，整个软件的强度不取决于最薄弱的一环，而是取决于所有薄弱环节的乘积。所以我们在设计架构的时候，要把精力均匀的分散开，避免某些类它异常的健壮，而有些类只是勉强的健壮。

第三点整个架构是否在概念上协调一致，这个很好理解，这种东一榔头西一棒子的设计肯定不是一个好设计。

第四点顶层设计是否独立于用作实现它的机器和语言，这个我们就必须要考虑编程语言所带来的影响。

举个例子，它有些语言它是保证多线程安全的，有些语言它是不能保证多线程安全的，因此我们在设计这个架构的时候，就不得不考虑这个资源所是怎么样运用。

第五点，是否说明了所有的主要动机。在架构设计的时候，我们经常可能有提出来了多种的方案，但是最后我们只会采用其中一种方案，我们就必须要在架构文档中详细的说明每一种方案以及他们的优劣势，还有我们最后做出选择的原因，为什么？因为在这种长周期的开发中，可能会临时更换方案，这个是不可避免的。

这时候我们很可能这回更换到了一个我们早已经否定了的方案，因为时间长了我们忘记了是吧？我们又开启了一个我们已经被我们否定了的方案，为了避免这种尴尬的事情发生，详细的记录所有主要决策的动机是必不可少的。

第六点，作为一名实现该系统的程序员，你是否对这个架构感觉良好？我认为经过良好设计优秀设计的系统，会让实现他的程序员为他感到骄傲， Ok。第三章的内容就到此全部结束了，希望你也能像我一样有所收获。